December 2013

Development of Android Applications

Angelia Rachel Giannone
Worcester Polytechnic Institute

Edison Jimenez
Worcester Polytechnic Institute

Kyle Peter Davidson
Worcester Polytechnic Institute

Tyler Orrin Morrow
Worcester Polytechnic Institute

WPI

Development of Android Applications

An Interactive Qualifying Project Report

Submitted to the Faculty of
Worcester Polytechnic Institute

In partial fulfillment of the requirements for the
Degree of Bachelor of Science

Submitted on December 16, 2013

Written by
Edison Jimenez
Kyle Davidson
Angelia Giannone
Tyler Morrow

Approved by
Professor Vance Wilson

Abstract

The goal of this project was to design small and robust Android applications that can be used by Worcester Polytechnic Institute to teach the concepts of Android programming. These applications and their corresponding instructional guides will focus on teaching the programming of screen navigation, decision logic, database interaction, and user interface controls.

Table of Contents

List of Figures

Chapter 1 - Introduction

Mobile applications are becoming increasingly prevalent today, particularly in the world of business. As such, many prospective business students are interested in creating mobile applications but lack the knowledge to do so. The goal of this project was to develop two Android applications and tutorials for novice programmers to build these applications for a future course at WPI. These two applications introduce many necessary concepts of Android application development and should be challenging but feasible for novice programmers to build. Students will learn basic programming principles along with the necessary skills to develop an Android application.

About this Guide

This guide serves as a tutorial for novice programmers to learn the basics of Android application development. It is highly recommended that students supplement the knowledge provided with their own research, particularly from the official Android Developers documentation.

Chapter 1 covers the IQP team's process in developing these applications and this tutorial. This includes weekly goals and accomplishments, evolution of the applications over time, why some concepts were included and others were scrapped, and the team's general thought process as the semester went on. Chapter 2 shows the final results of development in a chart format. Various skills were deemed necessary for this project, and the chart shows which applications help to build which skills in the students.

Chapters 3, 4, and 5 serve as instructional tutorials for developing the applications created by the team. Chapter 3, which works with the provided Stub App, helps students to set up their development environments and understand the basics of Eclipse and the different pieces that form an Android application. Chapters 4 and 5 walk the students through creating the QuizMe and HomeworkHelper applications.

Formatting and Terminology Notes

- An Android **Activity** is a single screen of an application, such as a main screen or the settings menu.
- **Inflating** an Activity means to load it into the current view of the application, bringing it's formatting and methods with it
- **Bold Text** in this guide is used primarily for programming controls, as well as for emphasis in some sections
- "Quoted Text" is literal text, typically used when asking students to input a single line of code or a particular file name or setting exactly as written.
- *Italics* are used to denote file names and directories
- Code is shown in Courier New font
- **Bold code** is code for students to add into their programs. This is not treated as literal text ("quotes") as it is typically formatted in writing as if it were seen in an IDE
  ~~Struck Through code~~ represents code that is currently in a student's program and should be deleted

Chapter 2 - The Initial Development Process and Reasoning

This IQP began with a meeting of the group and Professor Wilson in order to solidify the goal of the project. This goal was to produce course assignments for a future course at WPI, IT 270X, to teach the basics of programming Android applications. In these assignments, focus was to be given on implementing screen navigation, decision logic, database manipulation, IDE capabilities, and user interface controls. Over the next week, the group researched Android application development, reviewed available application development environments, and brainstormed application ideas to be used as programming assignments. The group's primary source of knowledge of Android development, as well as additional information on Android functionality, was the book *Android Programming: The Big Nerd Ranch Guide* by Bill Phillips and Brian Hardy. This book supplied the foundation for the group's Android applications through the use of miniature application examples. The book also influenced the groups selected development environment: the Android Developer Tools bundle, available from the official Android Developers website as a free download.

Week 1

The goals of the first week were focused around the logistics of the IQP, which included establishing meeting times and discussing documentation, finding and evaluating tools to develop applications, and determining the scope of the applications. When brainstorming application ideas, it was key that each potential application included most of the programming skills outlined previously.

Each group member was responsible for coming up with two to three ideas for applications and discussing their proposed functionality as well as how each application met the criteria presented in the design goals. The initial proposed applications were a GPS localization tool, a spreadsheet manager, a gas price and usage tracker, a mobile version of the WPI library's search engine, a battery monitoring application, a programmable study guide, a unit conversion application, a daily planner, and a personal web database application. These ideas were to be proposed and evaluated further at the next meeting.

Week 2

We decided to set the Eclipse bundle provided by the Android Developer's website as the recommended development environment for the class because it provides all the necessary tools for development in a single archive file. This archive can be easily downloaded, extracted, and included into the current version of Eclipse without any additional configuration. Application proposals were also presented and narrowed down by feasibility. The final three applications to be marked for further planning were the programmable quiz application (QuizMe), the daily planner application (HomeworkHelper), and the battery monitoring application (BatteryManager). The team then put together graphical mockups of what these applications should look like. Examples of our mockups are shown in Figure 2.1:

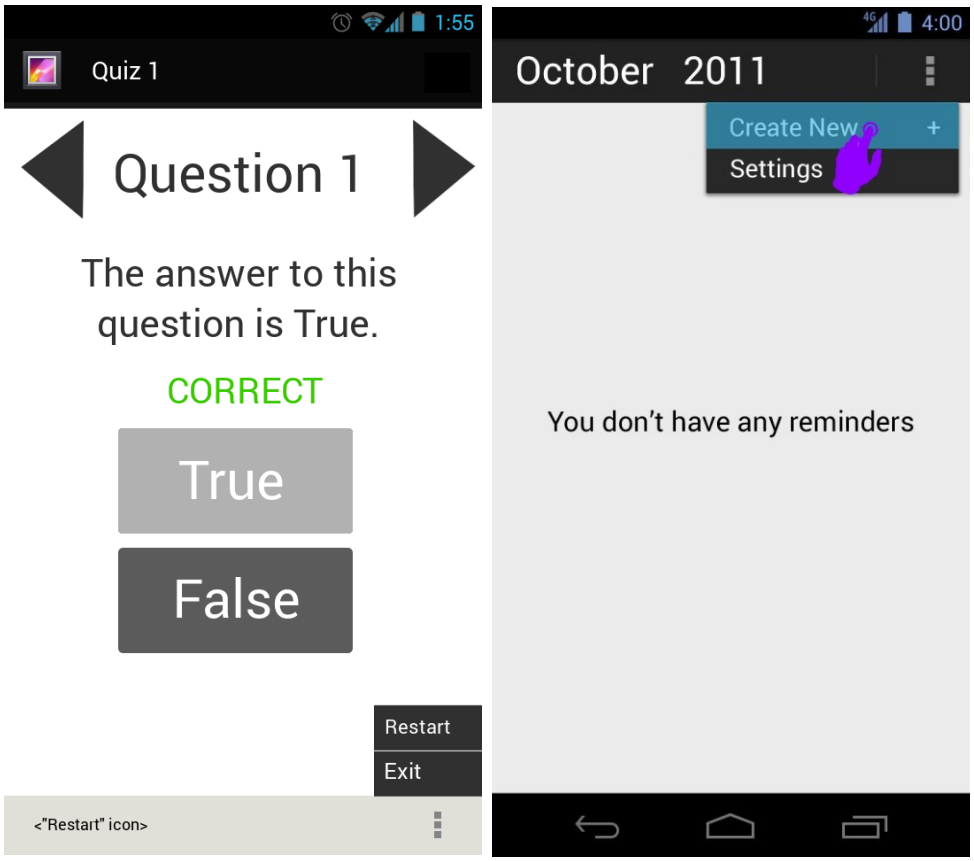

Figure 2.1: Initial mockups made in Photoshop prior to development

The team also estimated student completion times for each of these three applications. In terms of storing data, the decision was made to utilize SQL databases whose implementation would be provided to the students of IT270X. Flowchart behavior diagrams were also created for the QuizMe and HomeworkHelper applications to further analyze the development process and feasibility, as shown in Figure 2.2 and Figure 2.3 below:

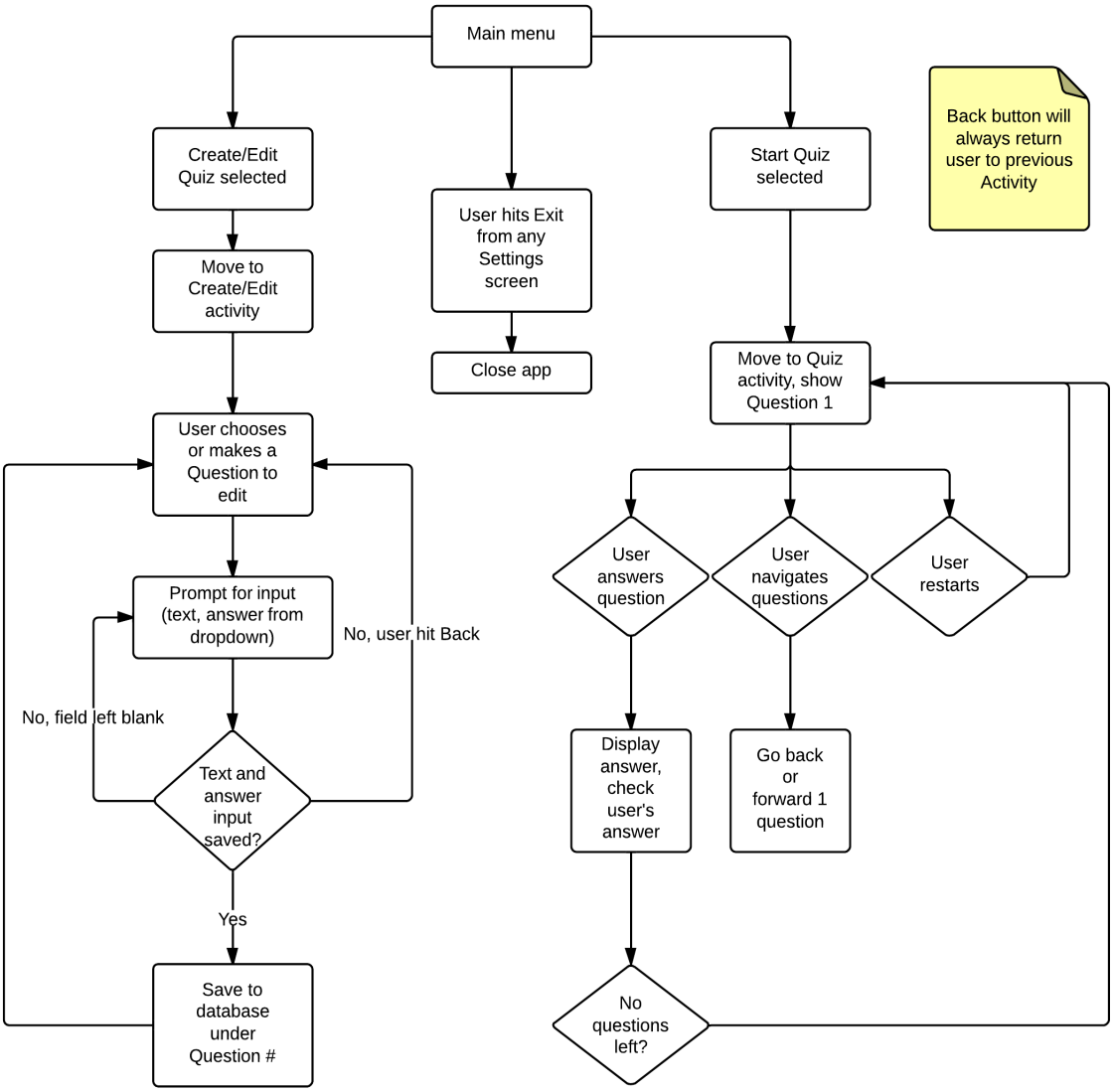

Figure 2.2: Initial flowchart design for the QuizMe application

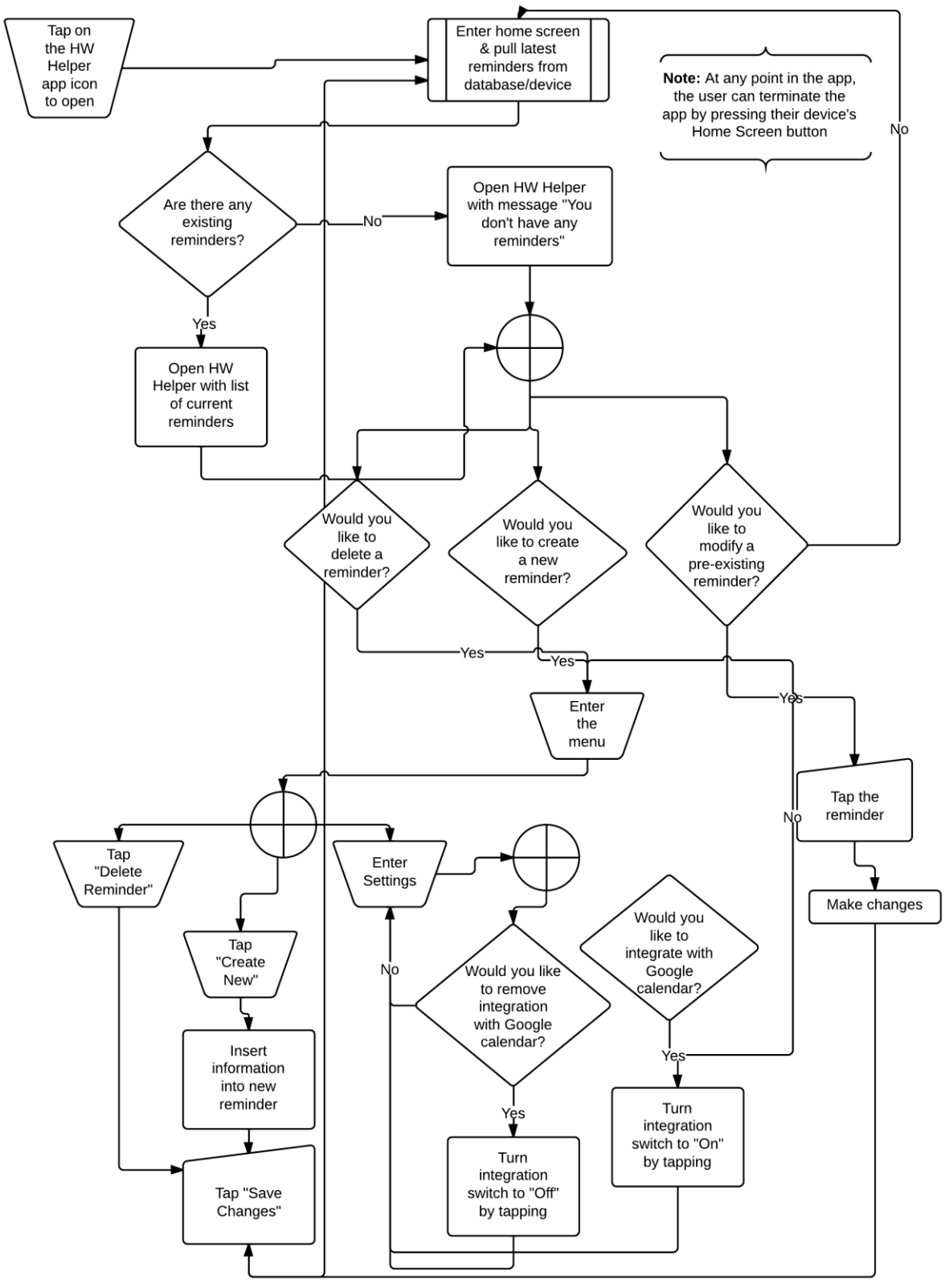

Figure 2.3: Initial flowchart design for the HomeworkHelper application

Week 3

During the third week, it was decided that development would begin on the QuizMe and HomeworkHelper applications. The Battery Manager application idea was set aside, only to be included in the development process if time allowed. The group decided to use pair programming methods for development with each pair of the group working on one of the two applications. Kyle Davidson and Tyler Morrow were assigned to the QuizMe application while Edison Jimenez and Angelia Giannone began developing the HomeworkHelper application.

The group decided to manage the code with some sort of source control repository in order to allow all members of the group to access the code of either project. This turned out to be highly beneficial, especially when a pair ran into a problem during development. The group set up a Git repository managed through GitHub for this purpose.

A third application proposal was added to the development goals. Professor Wilson wanted the team to develop a stub application that would demonstrate basic functionality that was common to both applications. The application was to contain an editable textbox as well as a button that would hide and reveal the textbox whenever the button was tapped on. This application will be explained in detail shortly as the development process of each application over the next four weeks is discussed below.

Weeks 4 – 7: Developing the QuizMe Application

The QuizMe application is a simple study helper application designed for students that is also easy for novice programmers to develop. This application implements basic SQLite database creation and manipulation to store both Quizzes and Questions associated with a given Quiz. QuizMe also introduces students to the concepts of Intent Extras, Fragments, various Button types, as well as essential programming skills such as decision logic and basic array manipulation.

It was decided at the start of development that each Activity of this application would be built in a hierarchy fashion for ease of programming for both the IQP team and the future students of this course. Each Activity would inflate one of the Activities below it when needed and return to the Activity above it when finished. This hierarchy method also allowed for the testing of near complete functionality as each piece of the application was created. Initially, the application was designed to handle True or False Questions, with other Question types to be added later as an expansion. Development began by creating an SQLite database class to store created Quizzes. By working off of a basic tutorial from AndroidHive (Tamadi), a database specific to the needs of this application was created and later modified.

Once the database classes were finished, the application needed an Activity to list the available Quizzes. This was accomplished by using an Android view called a ListView. This ListView was populated by using another Android object called a Cursor Adapter. Because of this, the database returned all Quizzes stored in it into an Android object called a Cursor. This Cursor Adapter method proved to be overly complicated for the purposes of this application and was not kept in the later part of the development process. In contrast, the HomeworkHelper application uses the Cursor Adapter method successfully, but QuizMe was meant to be the simpler of the two applications. Therefore, this design was amended to utilize another adapter type that manipulates an array of objects. This Array Adapter proved to be much easier to work with and was implemented successfully. The database returned a list of Quiz objects that were

then copied into an array to be used by the adapter, which populates the ListView. When an item from the ListView is tapped on, the SQL table ID of the corresponding Quiz will be passed through to the next Activity using Intents. Intents and Intent Extras allow information to be easily passed through Activities. The XML code for this Quiz List Activity was extremely simple and contained only a single ListView.

The next Activity developed allows the user to create new Quizzes. This Activity displays a textbox informing the user of purpose ('Create a new Quiz'), an EditText field for the user to enter the name of the Quiz, and two buttons, one to save the Quiz using a function defined in the SQLite database, and another to discard the Quiz being created. It was decided that the Quiz List Activity would automatically inflate this Activity if there were currently no Quizzes stored in the database.

Another Activity was needed in order to allow the user to run or edit a selected Quiz. Intent Extras were used to pass the SQL ID number of the Quiz being operated on from the Quiz List Activity into a key String of this Activity. Once the Quiz ID is known, the user can run, edit, or delete the selected Quiz without corrupting any other Quiz in the database. Quiz deletion was implemented first as it requires no additional activities and only the use of a simple database method. A Quiz could not be run without any Questions (a Toast popup informs the user of this if they do try to run an empty Quiz). Therefore an Activity to edit Quiz contents was required.

Before this next Activity could be created, the application would require a second SQL database class to handle Questions. This Question database helper class behaves similarly to that of the Quiz database except for a few key differences. This new database stores True or False Question objects, which consist of Question text, an answer, and the associated Quiz ID, and is able to retrieve a list of Questions that are associated with a specific Quiz ID.

The Question List Activity, as this **Edit Quiz** Activity would come to be known, was designed to be functionally equivalent to the Quiz List Activity, utilizing an Array Adapter to populate a ListView of True or False Question objects and inflating a Question editor method when a Question is clicked. If no Questions are associated with this Quiz ID, the user will be sent straight to the Question editor Activity to create a new Question. Clearly this would become an issue when new Question types are added, but during this time, the idea of multiple choice and other Question types was also discussed but shelved for the time being.

This Question editor, Add Question Activity, would have to store a new Question into the database, or update a Question in the database; a **new Question** flag is passed through an Intent Extra in order to indicate which situation the user is in so the application can respond appropriately. The latter case uses more Intent Extras to pass information in the Question to the Activity to pre-populate the Question text and answer fields. This Activity allows the user to specify the Question to be displayed as well and to select an answer from a dropdown menu. This menu is implemented through a Spinner, which creates a menu from a list of Strings found in *strings.xml*. The user can save the Question, adding a new Question to the database or updating an existing one, or discard changes if they wished. If this was a preexisting Question, the user also needs an option to delete it. This option was placed into the appropriately named Options Menu, opened through an onscreen icon on some Android devices and the Android menu button on all devices.

With the first two weeks of development nearly completed, a final Activity was still required in order to actually run a Quiz. An array of Questions with the given Quiz ID is generated in the same way as the Question List Activity, except this information is not put into a ListView. Instead, this Activity uses this array to display the Question text and to compare the user's answer with the answer stored in the database. The Question text changes with each Question of course, while the True and False selection Buttons and Next and Back navigation buttons are persistent in the UI. When the Next or Back buttons are pressed, the current array location is shifted in the appropriate direction and the views are updated to display the next or previous Question. This array wraps around as well; if the user tries to go past the last value of the array in either direction, they will be greeted by the Question from the opposite end of the array. With this Activity complete, and after a bit of XML cleanup to improve the UI, the application was completely functional and ready for demonstration.

The next two weeks of development were dedicated to implementing additional functionality as well as fine tuning previous functionality and cleaning up the UI further. It was decided that the application should handle multiple types of Questions, specifically True or False, Multiple Choice, and Fill in the Blank. As well, it must feature the WPI logo prominently, though this would be a minor addition.

In order to create new types of Questions, new classes and modifications to existing ones would be required. In order to develop these additions, it would be necessary to implement two new Question classes, refactor the original Question class, and bind all three Question classes using a Java Interface in order to provide important abstraction. The Question SQL database would need to be modified to include a table for each type of Question as well as create and update functions specific to each Question class. The method for retrieving all Questions associated with a quiz ID would need to be changed as well to scan all three tables and return a list of objects that implemented the Question interface.

Fill in the Blank and Multiple Choice Questions cannot use an editor Activity that only allows True or False answers of course. As well, the list of Questions now lists objects that implement the Question interface rather than a specific type of Question object. The interface requires each Question to have a method to prepare an editor and return the Intent for that editor. With this added, clicking on a displayed Question object, or one of the three newly implemented **Create New … Question** buttons in the Options menu, calls the **Prepare Editor** method defined by the interface, and the action to take is determined by the specific Question class. The application now utilizes a different Add Question Activity for each class and the **Create Editor** interface method causes each Question to pass an Intent to its specific editor, allowing the Question List Activity to create three different editors without type checking.

The new Question types are functionally similar to the original True or False class, but many changes had to be made. Fill in the Blank utilizes a second, smaller EditText rather than the Spinner, and the **Check Answer** method of this class (to be used while running the Quiz) simply performs a String comparison (ignoring case and stripping whitespace). Multiple Choice, however, is much more complicated, and will certainly be a crash course in decision logic for students. The UI for both the Multiple Choice editor and fragment (fragments will be discussed shortly) consists of the Question text field and four or less radio buttons, each with a text field displayed next to it. Android Radio Buttons must be contained within a Radio Group so that only one may be active at a time. However, this only affects what happens when the button is

pressed; visually, other buttons in the group do not automatically deselect. When a Radio Button in this application is pressed, the answer value is set appropriately, and the other buttons are all deselected by setting their built-in selection method to False. Various other checks are in place so that at least two answers are filled out, but to go in-depth on this topic would be redundant as the majority of Multiple Choice workings are built through groups of basic If-Else statements.

Running the Quiz requires significantly more work to implement three different types of Questions rather than just one. Unlike editing Questions, the Quiz runs inside a single Activity; as such, preparing an Intent elsewhere and passing it in was not an option, although it is a step in the right direction. This task is implemented by using Fragments. Without going too deep into the application code, a Fragment in Android development can be thought of as a miniature Activity running inside a full Activity. Both the XML layout and Java code are similar to that of a full Activity, with some key differences for fragment implementation. In the XML layout file for the Run Quiz Activity, everything regarding the Question display is replaced with a **fragment container**. A fragment is selected and displayed while the application is running, so a **Prepare Fragment** method is added to the interface and the three Question classes, allowing the Activity running the Quiz to get a Question from the array, find the proper fragment and display that fragment to the user with the data from the appropriate Question visible. The Next and Back buttons remain unchanged.

With this new functionality implemented, final adjustments would be made to improve the application as a whole. A new Activity was added, a Splash Screen that displays the WPI logo before moving on to the Quiz List Activity. After a bit of refactoring, this was made to be the Main Activity of this app. For comparison, HomeworkHelper utilizes a ‘loading’ style Splash Screen, while ours uses a Button to begin. As such, the HomeworkHelper Splash Screen is NOT the Main Activity of that application. Other adjustments to QuizMe mostly involved editing XML files in order to improve the user interface, such as replacing the **Delete** option in some menus with a trash can icon in the action bar, repositioning buttons, and resizing some layout objects. As well, the “CORRECT!” and “INCORRECT!” text now display in green or red respectively.

As mentioned, this application will be developed by novice programmers in a new course offered at WPI. The goal of this course is to teach the basics of Android application development rather than detailed programming. As such, it is important to provide some code to these students to work off of. Provided code will consist of two fully implemented database classes, along with three Question classes and one Quiz class, each of which will have enough implementation to function properly with the database, and an interface class to bind all three types of Questions together. Most methods will not be provided. Snippets of code will be given to the students in the instructions if necessary, particularly for fairly difficult concepts for beginners such as arrays and fragments. Of course, even if code is provided, students should still study it carefully and try to understand what is going on. The provided code is not meant to be copied and forgotten. Instead it should be worked off of and improved upon.

Weeks 4 – 7: Developing the HomeworkHelper Application

The HomeworkHelper application is a calendar-based tool designed for WPI students to create customizable reminders for upcoming assignments. This application allows the user to

create, modify, and delete multiple reminders, to be stored in a local SQL database. Creating the HomeworkHelper application allows students to dive into an in-depth approach of the Android Development Tools, decision logic, and database and array manipulation.

The first three weeks of application development consisted of the team becoming accustomed to the ADT environment, solidifying the intended functionality of the application and creating graphical mockups using Adobe Photoshop, which showed the step-by-step user experience of the intended HomeworkHelper application as well as provided the outline for how the application would function. This provided a great template for the developers as it solidified exactly what the final application would look like. Once the application structure was decided upon, the remaining three weeks were dedicated to developing the application in Eclipse. The first step was to create the source file *MainActivity*, the menu items, and the *SettingsActivity* file along with their respective XML files.

The Main Activity displays a list of upcoming reminders, if any exist, which are stored in a local SQLite database. The Main Activity also implements a menu button that consists of switch case logic statements that display the options to either create a new reminder or go to the Settings Activity.

Next, the Settings Activity and corresponding preferences XML files were developed. The preferences XML file creates the list of Settings options, originally 'Integrate with Google Calendar' and 'Enable Notifications.' These options were created using ADT **check-box preferences** whose values are either 'True' or 'False', based on the user having checked or unchecked the setting. After extensive research on how to integrate reminders with Google Calendar using the Google Application Programming Interface (API), the team collectively decided that there is too much overhead and too many technicalities associated with linking HomeworkHelper to a Google account and importing information from Google Calendar into reminders. Therefore, the team decided on integrating the calendar on the device instead.

The *EventActivity* was then created as this was how a user was going to interact with the application. Initially, the *EventActivity* was designed to contain a series of TextViews, EditTexts, a Spinner, and a Button. The following week, the TimePicker and DatePicker fragments were implemented. These **pickers** are predefined Android controls that allow the user to choose each part of the time (AM/PM, hours and minutes) and date (year, month, date) from a predefined scrolling selection. TimePicker and DatePicker are especially valuable for HomeworkHelper because they are universally formatted. You can find these in many other applications, including the default Calendar application provided by Android 4.0 and above. The DatePicker and TimePicker fragments defaulted to the current date and time, if the user does not modify the selection. These fragments also implemented decision logic in order to decide whether to default to AM or PM. Arrays are created and modified to store the time and date values, and are then returned as DatePicker Dialogs and TimePicker Dialogs, respectively. **Dialogs** are predefined Android objects, necessary for manipulating the time and date pickers. Later in development, calendar permissions were added to the *AndroidManifest.xml* file, allowing the user to have read/write capabilities to the device's internal calendar (or that of the emulator).

At this point, it was decided that the Android **Action Bar** should be utilized in this application. This was not necessary in QuizMe, but the functionality it can provide is excellent for HomeworkHelper. Action bars are similar to headers for each Activity in the application.

HomeworkHelper has an action bar in the Main Activity, inflating the menu and the application name in a text box. This action bar was implemented during the subsequent week.

Professor Wilson suggested the application should have a pre-populated list for the 'course' field selection to maintain the formatting integrity of application and keeping the application somewhat WPI related, and also modify the 'course' field to have searchable values. While this feature could certainly be expanded on, this was deemed enough for this point in time. This was created by adding a **Spinner** of predefined courses. The Spinner class is also a pre-defined Android class whose purpose is to accept an array of strings and create a drop-down menu of the string selection. The Event Activity was then modified to incorporate the **AutoCompleteTextView** Android adapter to allow the 'course' field spinner to update its list based on comparing substrings which the user inputs to the courses field. This provides the 'course' field to have searchable items.

During the final weeks of development, the HomeworkHelper database adapter source file was added, which creates and updates reminders from the SQLite database. Reminders are fetched using a **Cursor**. The Cursor class is a predefined Android class, used to write and access database information. When the user creates a new reminder, input strings and values are stored in corresponding rows in the database, and each row is given a unique row identification key to be referenced during read-write operations. To remove a reminder, the corresponding row is removed from the database. If there are no reminders, there is no database and so when the first reminder is created, a new database is made. Retrieving reminders from the database is performed using a cursor query. If the reminder is not found, an SQLite exception is thrown.

A new requirement of both applications added at this point was to include the WPI logo somewhere in the application. Similarly to QuizMe, this was added in a Splash Screen seen when the application is first launched. The Splash Screen is a feature that can be enabled or disabled in the Settings menu by toggling a checkbox. When enabled, the Splash Screen displays for a limited amount of time when the application is opened. After that time runs out, an Intent navigates the application to the Main Activity, where reminders are displayed, and a **finish** function is called which ultimately terminates the Splash Screen. If the Splash Screen is disabled in the settings, the finish function will run, without displaying the Splash Screen, and an Intent will be created, navigating to the Main Activity screen.

A discard button was added to the Event Activity so that a user could easily remove a reminder. When the delete icon is pressed, a dialog box pops up prompting the user if they want to delete the reminder. If the user chooses to delete the reminder, the corresponding row reminder is accessed, using the row identification key, and deleted. This two-step process, seen in many programs, helps to prevent accidental deletion.

Error handling logic was added in the Event Activity to handle the case of when a field is left blank when creating a new reminder. This was achieved by calling the predefined **setError** Android function on the specific field. If the length of the field is zero, an error is thrown and a red 'x' icon is displayed at the end of the field input area. The next step was to sort the items in the ListView and to categorize them in some way to allow for an easier flow of information and implement a higher standard for a user interface. Once this was completed, the final step of the application was to allow notification and calendar support depending on the settings the user applied in the HomeworkHelper application.

The next step was to allow users to be able to add reminders to their calendars and to allow our application to send out notifications. This was probably one of the more difficult things to implement because our team needed to figure out the *right* way to do certain things. At first, we wanted our application to be able to make new or edit existing calendar events without needing to bring up any other application. The amount of time and coding it would have taken to do this reached outside the scope of our project. Therefore, we ended up letting our application use the existing Calendar application on every Android device so that we could *pass* information to it. We then configured this so that a student could enable/disable this feature in the Settings Activity.

The final step took the most time out of the entire application. We needed a way to allow multiple notifications to be displayed based off of the reminders that a student sets in the application. Our first approach was to have the application read all existing reminders and set up notifications at every startup. Due to the inefficiency of this, we changed the approach to having a notification to be set up immediately after a student inputs the data. Using something called an **Intent Service**, which allows a portion of an application to run in the background without having to be open, we were successfully able to integrate this feature onto our application and allow students to enable/disable it in the Settings Activity.

Alpha-Testing

Once the applications and their respective tutorials were completed, the alpha testing process began. One volunteer from each development pair within the team would build the other pair's application from the instructions provided. This also includes setting up the Android ADT, importing the Stub App, setting up the emulator, and managing the Android SDK. Each volunteer would be going through the development process with the mindset of a novice developer, leaving criticisms and suggestions for improvement as such. The critiques and comments from this testing phase allowed the team to make necessary modifications to the tutorials to provide a more thorough and more educational experience for prospective students.

When testing HomeworkHelper, it was found that the tutorial left a lot to the students. Building the first half of the application was easy following the instructions provided, and the explanations as to how and why the application is being developed as such were clear and understandable. However, the second half or so is when much more was being left for the students to figure out. This is not necessarily a bad thing, as the students will not learn just by copying and pasting, but the explanations provided were minimal and the instructions said to "do this" or "do that" without explaining how. As well, several concepts were mentioned and utilized but not fully explained, such as the Action Bar. As the majority of this IQP team are computer science majors, it is no surprise that some assumptions were made that should not have been; this shows in the QuizMe alpha testing as well.

During Alpha testing for Quiz Me, it was found that the tutorial made a fairly smooth transition from explaining provided code to providing instructions for the students. Most concepts were well-explained, but sometimes the explanations were not necessarily to-the-point or they were grouped together in large code blocks, as opposed to being separated and then explained. This project also may also seem somewhat complicated to novice users, with respect to the many files being created and modified, whereas HomeworkHelper groups code into fewer

files. This may be beneficial for the student, however, as they are becoming acclimated with creating and modifying application activities and various other files.

Once adjustments were made to both tutorials, as well as any necessary changes to provided code, the guide would be submitted for beta testing.

Beta-Testing

After the completion of the alpha testing phase, the beta testing phase began, in which the tutorials and starter code were handed off to Professor Wilson for review. Adjustments from beta testing include further refinement in consistency as well as an alteration to the instructional documentation so that the Quiz Application could be built by modifying the Stub App. A small section was added to the beginning of the HomeworkHelper instructional to explain why the Stub App is not being utilized for that app. More figures were added throughout the document, and final formatting adjustments were made. Due to the significant changes made, alpha testing was performed again on the edited areas and changes to the tutorials were made accordingly.

Reflections

This project introduced the team to many new concepts. Android application development was a new experience for most of the group and should prove to be a useful skill in the future. The development process seems daunting at first, but Android development is primarily Java programming; it is not much different than most software development. The main deliverable of this project was this tutorial, not each of the applications we developed. Teaching new development concepts to those already familiar with computer science is not too difficult, but writing programming tutorials for novices was a new experience. It had to be assumed that students would have very little to no prior experience, so even simple concepts such as loops and if…else statements had to be explained. The team found that it was very easy to overlook concepts that would be common knowledge for experienced programmers. The alpha testing phase brought to light the coding practices that were not explained fully on the first draft, and beta testing brought up several more. As a result of this testing phase, more details were added to the tutorials as need be. In the end, the team is confident that these tutorials can be used to effectively teach Android development to novices. The skills gained through this project will undoubtedly be useful for the team in the future.

Application Comparisons

In order to build effective demonstration applications that teach the basic abilities and standards of Android application development and programming in general, a list of core values was put together. While some of these were required by the project, such as decision logic and database interaction, others were added by the team based on careful evaluation. The two application designs chosen from our initial list of ideas are two that meet the majority of these requirements. QuizMe will be developed first as it incorporates simpler usage of most requirements, while HomeworkHelper expands on these requirements and introduces several more, such as notifications and data integration with other applications. The core values are laid out in figure 2.4 below.

Learning Objectives	Stub App	QuizMe	HomeworkHelper
Screen Navigation		✓	✓
Decision Logic	✓	✓	✓
Database Interaction		✓	✓
User Interface Controls	✓	✓	✓
Use of Android Emulator	✓	✓	✓
Notifications			✓
Passing info to other applications			✓

Figure 2.4: Feature comparisons across the applications:

Screen Navigation: Students will learn how to enable a user to navigate between activities and menus in an app, as well as pass information between activities and menus. Students will also learn to use fragments to enable multiple displays on the same Activity.

Decision Logic: Students will learn the fundamentals of programming logic, such as **if** statements, **for** and **while** loops, and **arrays**. During this project, students will learn how and when to utilize these logic controllers in order to develop an application that meets the project requirements.

Database Interaction: Students will learn how to create and manipulate an SQLite database within an Android application. Databases will be manipulated by user input, which is then used in functions to add, edit, and delete entries from a database.

Interface Controls: Students will learn how to create a graphical user interface of an Android application through XML code and a graphical layout editor.

Android Emulator: Students will learn to use the Android operating system emulator provided with the Android SDK in order to properly test their applications during the development process.

Notifications: Students will learn how to run an application in the background and allow it to send push notifications to the Android status bar.

Passing Information: Students will learn how to integrate an application with other applications on an Android device so that information is shared between the applications.

Chapter 3 - Stub App

Overview

We will begin by setting up a Stub App, which has been provided in its entirety. The stub application will be used to initially set up your Eclipse environment and demonstrate the most basic aspects of Android application development. In addition, the stub application will serve as a convenient reference for your first buttons and text boxes.

Setting Up Your Workspace

The Android Developers web site provides a set of tools for application development, particularly a customized version of the Eclipse IDE with various necessary plugins. An existing Eclipse installation can be updated for application development, but it is not recommended for this project and we will not going over the details of this setup.

To begin this project, acquire the Android Developer Tools bundle from http://developer.Android.com/sdk/index.html. The ADT is provided for Windows, Mac OS X, and Linux, the instructions provided assume the programmer is using Windows. OS X and Linux users should be able to follow along without much difficulty regardless (on a side note, the applications you will be building were actually developed by the IQP team on all three of these operating systems).

Extract the downloaded file to whatever location on your computer you would like your developer environment to be located. Remember this location, as it will be important throughout this project. Next, acquire the stub application file, *StubApp.zip*, from your instructor and extract this file to the location of your choice (in this example it has been extracted to a folder called *Stub*). It is recommended to create a folder for this course, but ensure that this folder is not the same as your Eclipse workspace directory. This can cause a number of strange issues**.**

Start Eclipse from the extracted ADT bundle and select your workspace (you may wish to keep the default). Then select **File** → **Import**. This menu provides various code and project import options. Expand the **Android** menu and select **Existing Android Code into Workspace** (Figure 3.1).

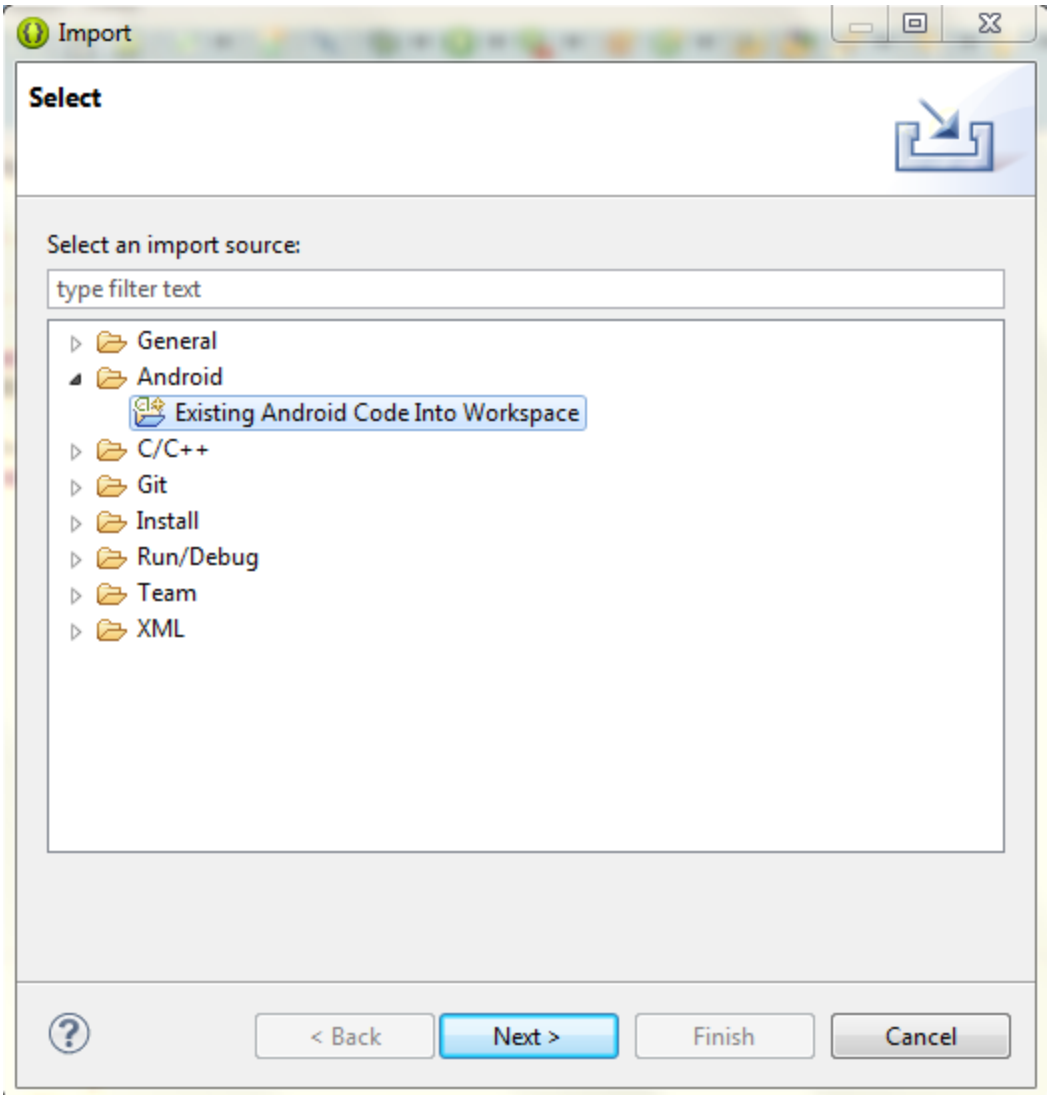

Figure 3.1: The Eclipse Import menu

Click **Next**. You will then be presented with another menu asking where the Android code is located. Select **Browse** and navigate to ***StubApp*** or the equivalent folder you created where you extracted the Stub App files and click **OK**. Make sure Stub App is the only project selected and ensure that "**Copy projects into workspace**" is selected (if this option is not selected, you may encounter some odd Android SDK errors). Click **Finish** to import the Stub App project (Figure 3.2).

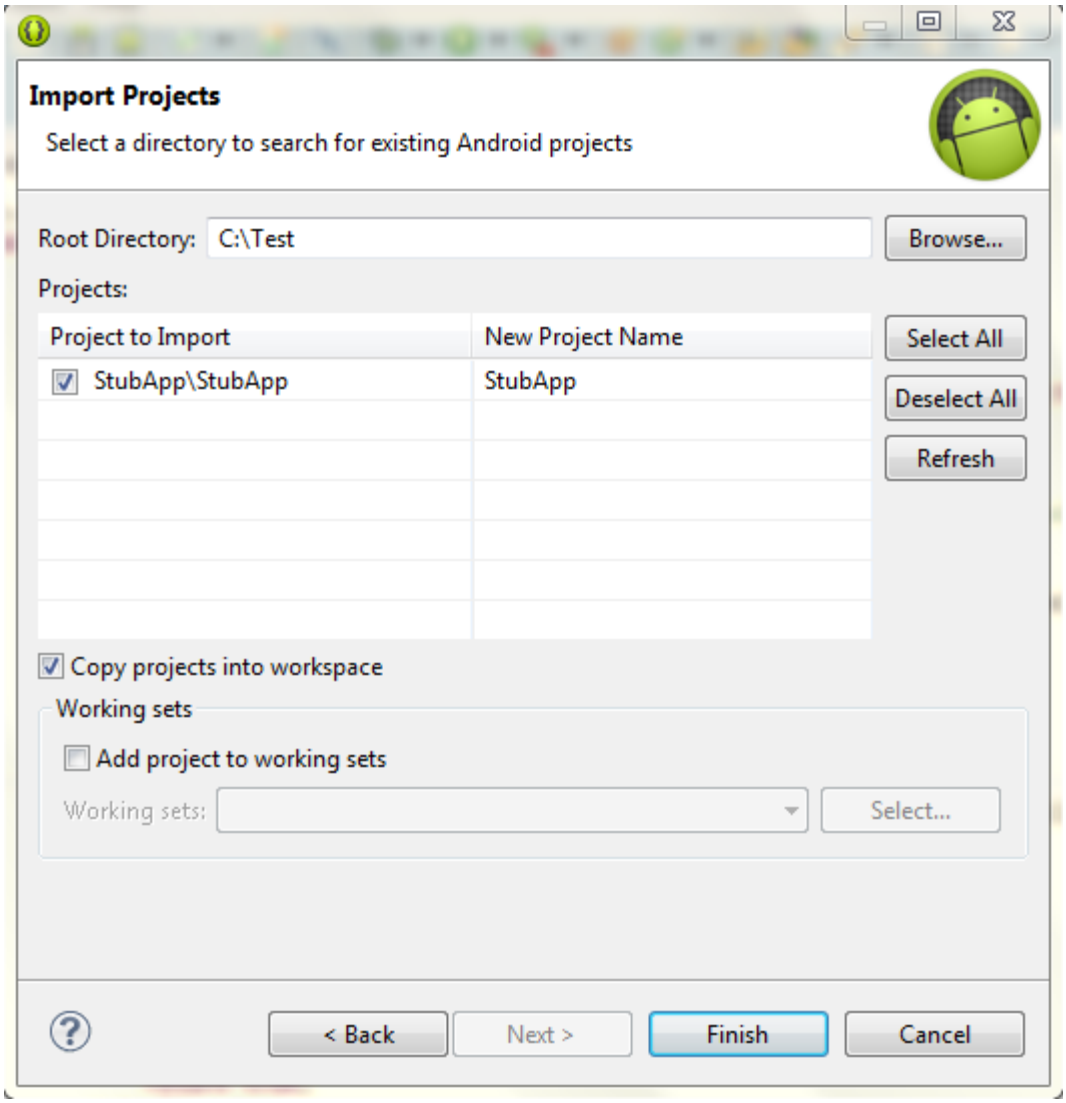

Figure 3.2: Import options

You may encounter an error such as “Unable to resolve target 'android-18'.” If you see an error such as this when importing a project, you are missing necessary Android SDK files and tools. In the main Eclipse window, select Window at the top, then Android SDK Manager. Select the Android SDKs you are missing (we recommend at least choosing any involving Android 4.0 and above, as well as any Android 2.0 version). Accept the licenses and download these SDKs. Next, back in the main Eclipse window, right-click on the *Stub App* project and click Properties. Select Android on the left of the window that appears and select the appropriate APK level (18) if it appears. Click Apply and return to the main Eclipse window. You may need to open and close the SDK manager again in order to refresh these changes. These errors should now be resolved. If not, please contact your instructor.

A Brief Overview of Eclipse

Eclipse is an IDE (integrated development environment) which supports many popular programming languages, though it tends to be very popular for Java programming. Android applications utilize XML layouts for the GUI (graphical user interface) and Java code for functionality. Also, the modified version of Eclipse provided in the ADT provides many additional functions for application development, such as creating a new Activity (essentially a single screen in an app), testing your application through an Android emulator, and quick and easy importing of necessary Android class libraries.

Setting up the Android Emulator

The Android ADT provides an emulator of the Android operating system. Applications in development can be installed onto this virtual operating system and tested as if they were being used on an actual device. All testing should be performed on this emulator first and foremost.

To set up the emulator, select **Window→Android Virtual Device Manager** from the main Eclipse window. In the window that appears, click New. In this new window, set up your virtual device as follows:

- **AVD Name**: This is up to you. Enter a short but descriptive name such as "GalaxyNexus."
- **Device**: Galaxy Nexus
- **Target**: Choose the highest API level you can. At the time of this writing, API level 19 (Android 4.4) is available. These applications were initially developed with API 18 (Android 4.3).
- **RAM**: 512 on Windows, 1024 on Linux or OS X
- **Internal Storage:** 200 MiB

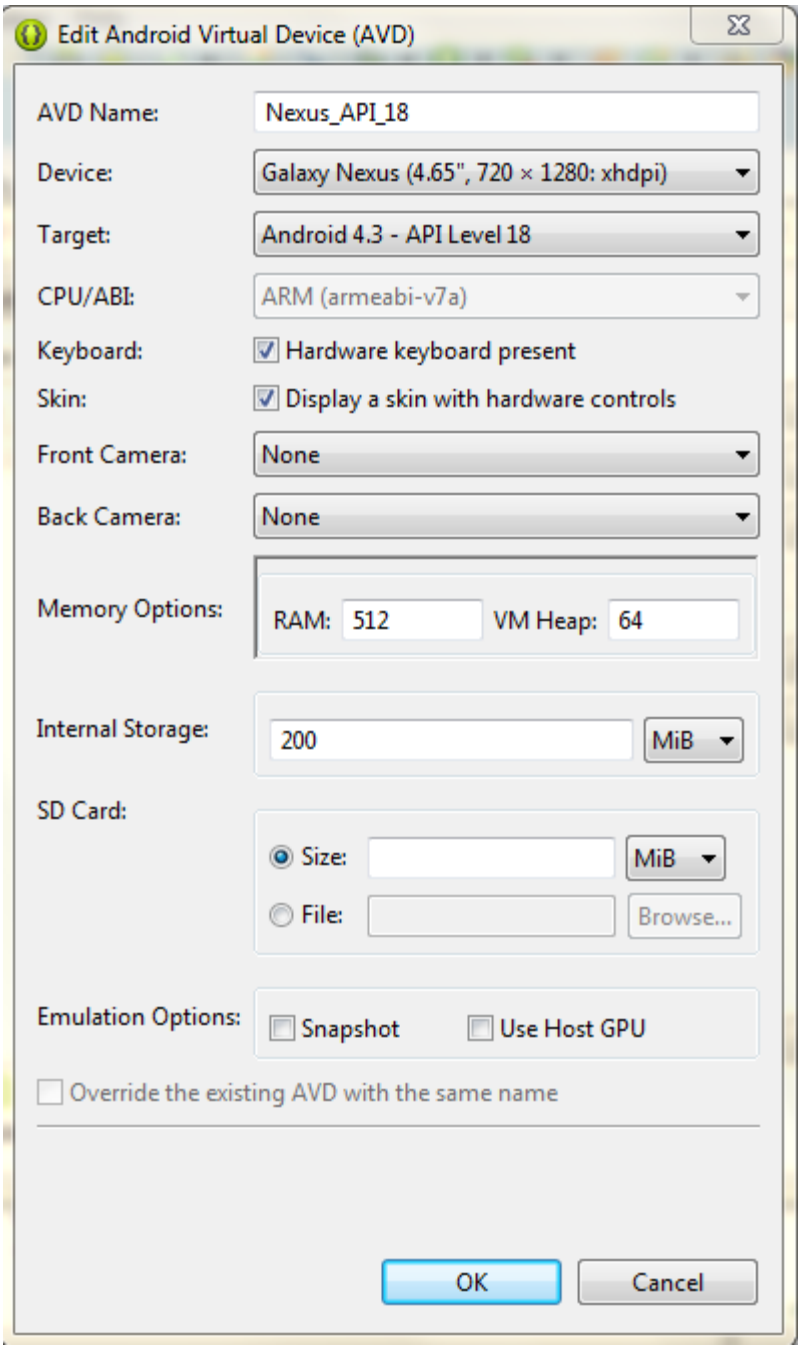

Figure 3.3: Creating a Virtual Device

Leave the other settings at their default values. Once your settings match Figure 3.3, click **OK** to save your emulator settings and the resulting window appears the same as Figure 3.4, close out the Android Virtual Device Manager.

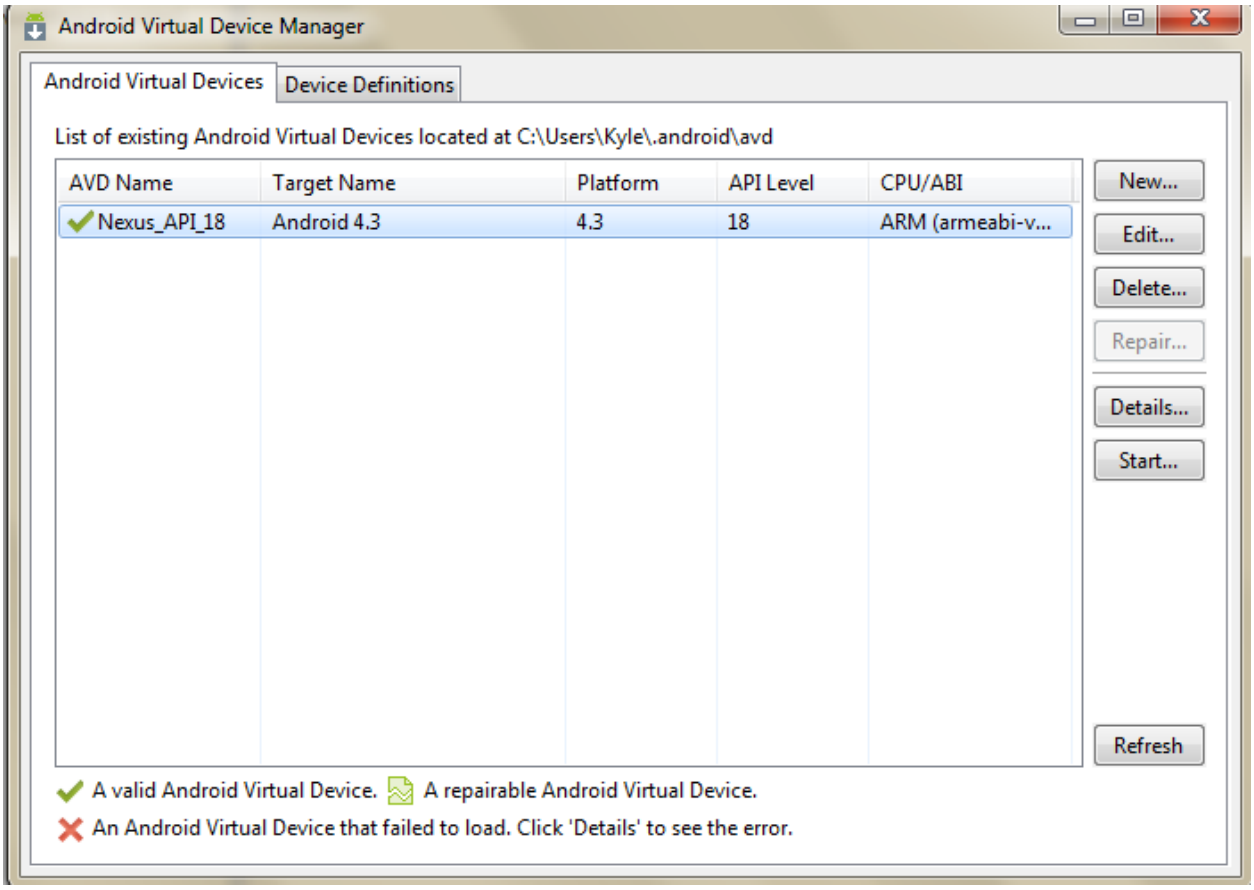

Figure 3.4: The Android Virtual Device Manager in the ADT

Testing on an Android Device

It is not necessary for you to own an Android device in order to test Android applications, and it is even less necessary to test on your own device. Still, testing on an actual Android phone or tablet may provide further insight into what needs to be adjusted in your applications.

For instructions on how to run Android applications on an actual device, please refer to the Android Developers guide on the subject, as the instructions vary by Android OS version, the device itself, and the current ROM running on your phone or tablet.

http://developer.Android.com/tools/device.html

This task may seem daunting, so please keep in mind that it is not necessary for this project. The emulator will suffice.

Keep in mind, however, that for professional development, you will always want to test on an actual device as well. In addition to testing on an actual device, you would want to test on multiple virtual devices as well through the emulator.

Running the Stub App

Figure 3.5: The Stub App

Now that your testing environment is set up, right-click on the Stub App project *Stub App* in the Project Explorer panel on the left side of Eclipse and select **Run As → Android Application**. This will launch the emulator and install and run the application. The emulator will take some time to start up, especially on the first time you launch it (on older hardware, this may take upwards of ten minutes). Please wait for the emulator to load. (If you are testing on an actual Android device, as long as it is plugged in via USB, the emulator will not be launched, but the application will instead be installed and automatically run on your device). Once the application has loaded, play with it. Fill in some text and press the button a few times. Your development environment should be completely set up. As well, the stub application utilizes a few key components of application development in terms of both the GUI and the underlying code.

If you find that the current emulator is too large for your screen, or the performance is too slow to work with, open the virtual device manager and select a different device instead of Galaxy Nexus. Most devices will work fine for this project, the Galaxy Nexus was just chosen for

consistency purposes. If you encounter any errors when launching the app, such as “Failed to install APK,” with the emulator still open, attempt to run the application from Eclipse again.

In Android development it is important to create the layout of an Activity before the actual code, as the layout code will be referenced for functionality. To view the XML code that controls what you see on the screen navigate to the *res→layout* folder in the Project Explorer (on the left side of the Eclipse window) and double-click *activity_main.xml*. This will open a graphical layout tool (which will not be used often). At the bottom of the screen you will see the options **Graphical Layout** and *activity_main.xml*. Click on *activity_main.xml* to see the actual code controlling this layout:

Once the XML code is on screen, you will notice it behaves like a nested list, with items declared with < and ended with />. The first thing to note is the `<Linear Layout` at the top (ignore the `xmlns:android` field as that will always be attached to the outermost list item). It is worth noting that all the individual fields in XML can be distilled into label value pairs, for example, android:orientation is a label and “vertical” is the text value that is assigned to the label in the code shown below. The Linear Layout setting informs Android that items will be displayed one after the other. The Linear Layout has four fields:

```
android:layout_width="match_parent"
android:layout_height="match_parent"
android:gravity="center"
android:orientation="vertical" >
```

The layout width and height fields tell the object how to fill the space it is given. In the case above, `match_parent` tells it to match its parents dimensions; in this case, the parent is the screen itself. Another option for width and height is `wrap_content`, which will effectively wrap around if it runs off the screen. The gravity field tells the program where to display objects. A gravity of `top` will cause the layout to display an object as far up on the screen as possible, then start working downwards, `center` informs the code to keep objects as close to the center as possible. The orientation `vertical` tells the code to display objects top-to-bottom whereas `horizontal` would display left-to-right.

Next of note is the `>` at the end of `android:orientation="vertical" >` . This tells the code that the fields controlling the Linear Layout have ended and what follows are objects inside it. In this case two fields exist inside the layout, an EditText and a Button. These objects have height and width fields like the Linear Layout. New fields to take note of are the `android:id`, `android:text`, and `android:hint` fields. The `android:id` controls the name by which this object will be referenced in Java code. For example, `android:id=”@+id/textbox”` will cause the EditText object to be referenced by the name “textbox” in the application’s Java code. The hint and text fields behave similarly, taking a string from a file called *strings.xml* and displaying it. The “hint” value is used by the EditText to display a hint that will disappear when the user types in the textbox, while the “text” value will always be present on the Button. You will notice that this code references a string using `@string/button`, the `@string/` tells the program to look in the *strings.xml* file while the “`button`” part denotes the name of the string. Both the EditText and the Button are ended with `/>`. This is because they have no nested objects to manage. The last thing in the file is

`</LinearLayout>`. This indicates the end of objects managed by the Linear Layout. A similar ending is always required when ending an XML file with nested objects in it.

The final thing to look at before moving on to Java code is the *strings.xml* file (*res→values→ strings.xml*). This is where the text strings referenced by the Button and the EditText are stored. Each string is defined by a line following the format:

```
<string name="button">This is a BUTTON</string>
```

The only fields you need to worry about changing are `button` which gives the name the string is associated with and "`This is a BUTTON`" which is the string that is displayed.

Now, let's look at the actual code for this application. In the variable fields of the Main Activity class (*src→edu.wpi.it270x.Stub App→ MainActivity.java*) you will see the field `EditText textbox;`, this is an example of an imported class and has a corresponding `import Android.widget.EditText;` in the expandable list of imports. This list initially appears minimized as ⊕ `import android.os.Bundle;`, but clicking the plus icon will reveal all imports. The EditText class is an editable textbox that you can type inside while the application is running. You will also see a method:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    …
```

This is the code that is run when the application starts. The main thing to notice is when the EditText and Button are defined is the findViewById function. This process uses an automatically generated file called *R.java* to return an object linked to your XML code. In the above section the XML code identifies the EditText object with `android:id="@+id/textbox"`, this means that in the Java code it is linked with the function `(EditText) findViewById(R.id.textbox)` the EditText in parenthesis is a necessary cast when retrieving XML objects (casting is not something we will be using much in this project, but in short, a cast tells the program to read the information following it as a different type. For example, this code is reading the ID as an EditText. You will notice that the textbox field is not defined initially. This is because it is not referred to inside a local function; this will be discussed later. The final thing to note in the Stub App's code is the Button's onclick listener:

```
button.setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View arg0) {
            …
```

This causes whatever is written inside the onClick function to be executed whenever the button is clicked. In the case of the Stub App, a boolean variable called `status` is flipped, causing the EditText (a place where the application user types words) on the screen to become

invisible or visible. A boolean is a value that holds either True or False. In this case, this boolean value is checked in an if statement which runs a chunk of code when `status` is True, and another chunk of code is run when `status` is False. We will cover this more within the application tutorials.

Chapter 4 - QuizMe (Instructional)

We will now begin the QuizMe project. This application is a basic study guide. You can create a quiz and populate it with Questions and Answers. When people take the quiz, they will receive immediate feedback after giving an answer.

4.1: Project Setup

To set up your development environment to begin working on QuizMe you will modify your Stub App. The first thing to do is change the project's name. To do this right-click on the Stub App folder and select **refactor→rename** as shown in Figure 4.1. Rename the project to "QuizMe" (Figure 4.2).

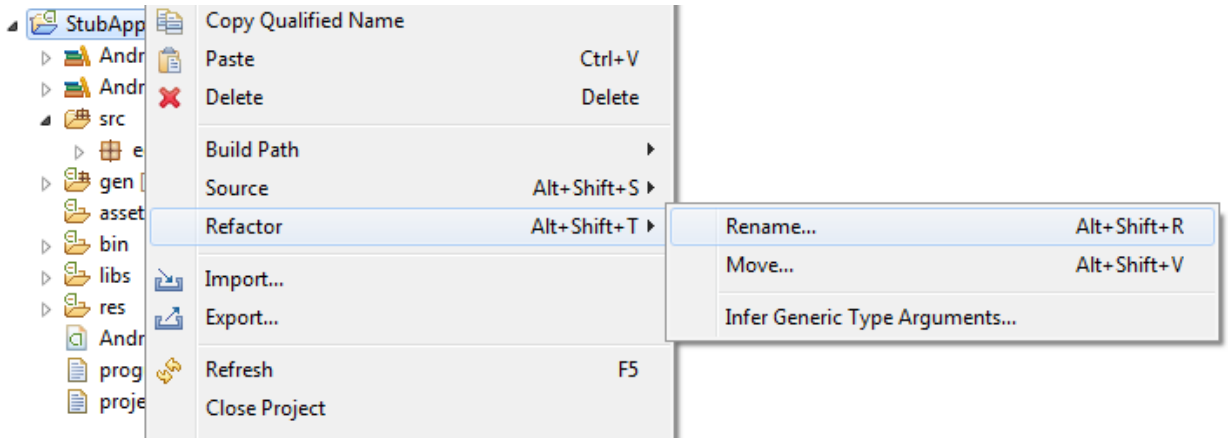

Figure 4.1: Refactoring

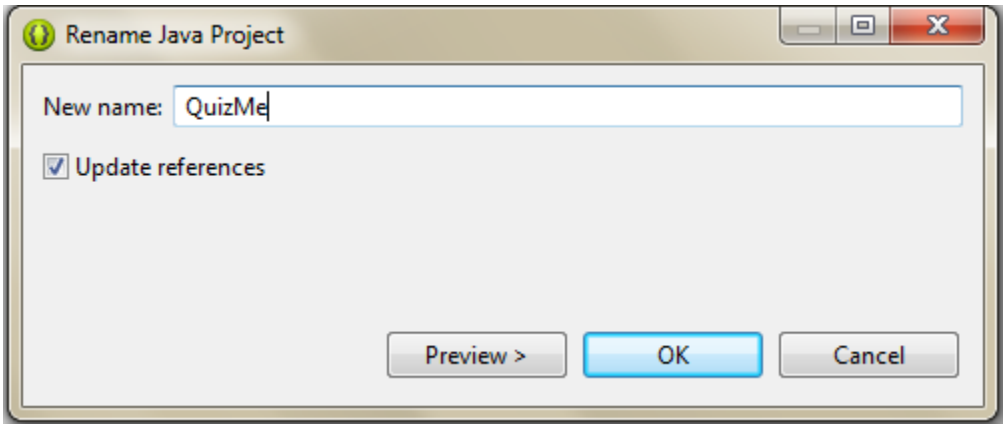

Figure 4.2: Renaming

Now rename the package inside the *src* folder to "edu.wpi.it270x.quizme" via the same refactoring method (Figure 4.3). This will rename the entire project.

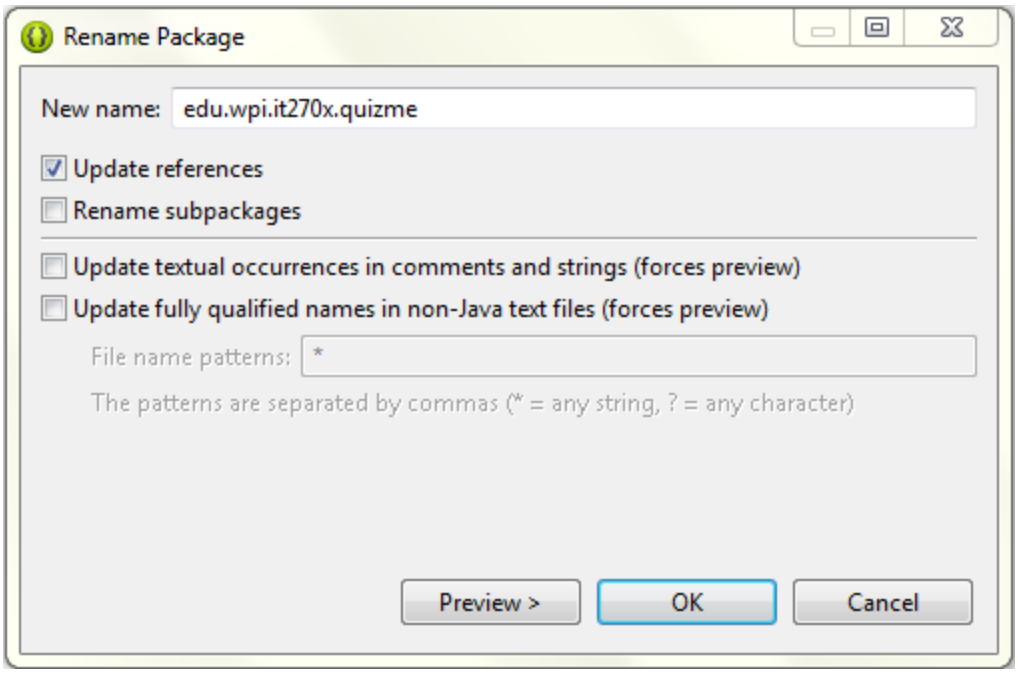

Figure 4.3: Renaming Packages

Next you must import code necessary for your project. Your instructor should have provided you with a zip file *quizme_provided_material.zip*. Extract this file to a directory that you will remember, preferably one designated for your application development projects. Right-click on the newly renamed package, *edu.wpi.it270x.quizme*, and select **Import** (Figure 4.4).

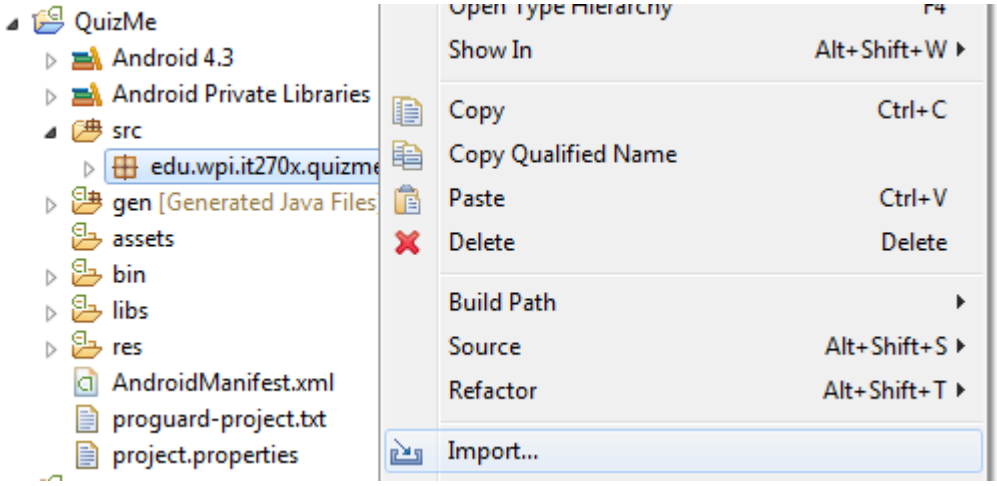

Figure 4.4: Importing Code

In the next menu, expand the General category select **File System** (see Figure 4.5).

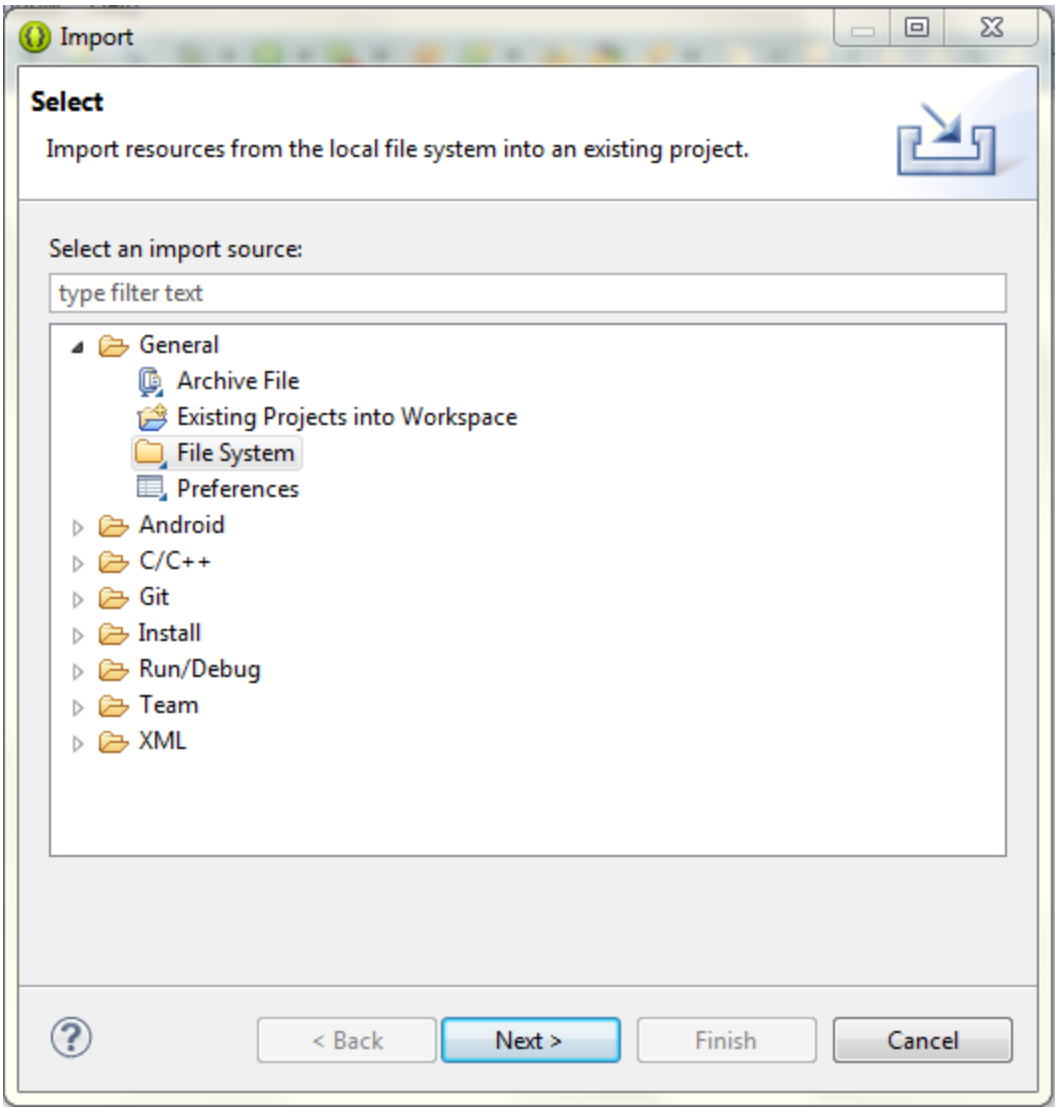

Figure 4.5: The File System Import dialog

Click **Next**. In the dialog that appears, click Browse and navigate to the directory in which you extracted *quizme_provided_material.zip* and select the *code* folder inside the provided materials folder (Figure 4.6).

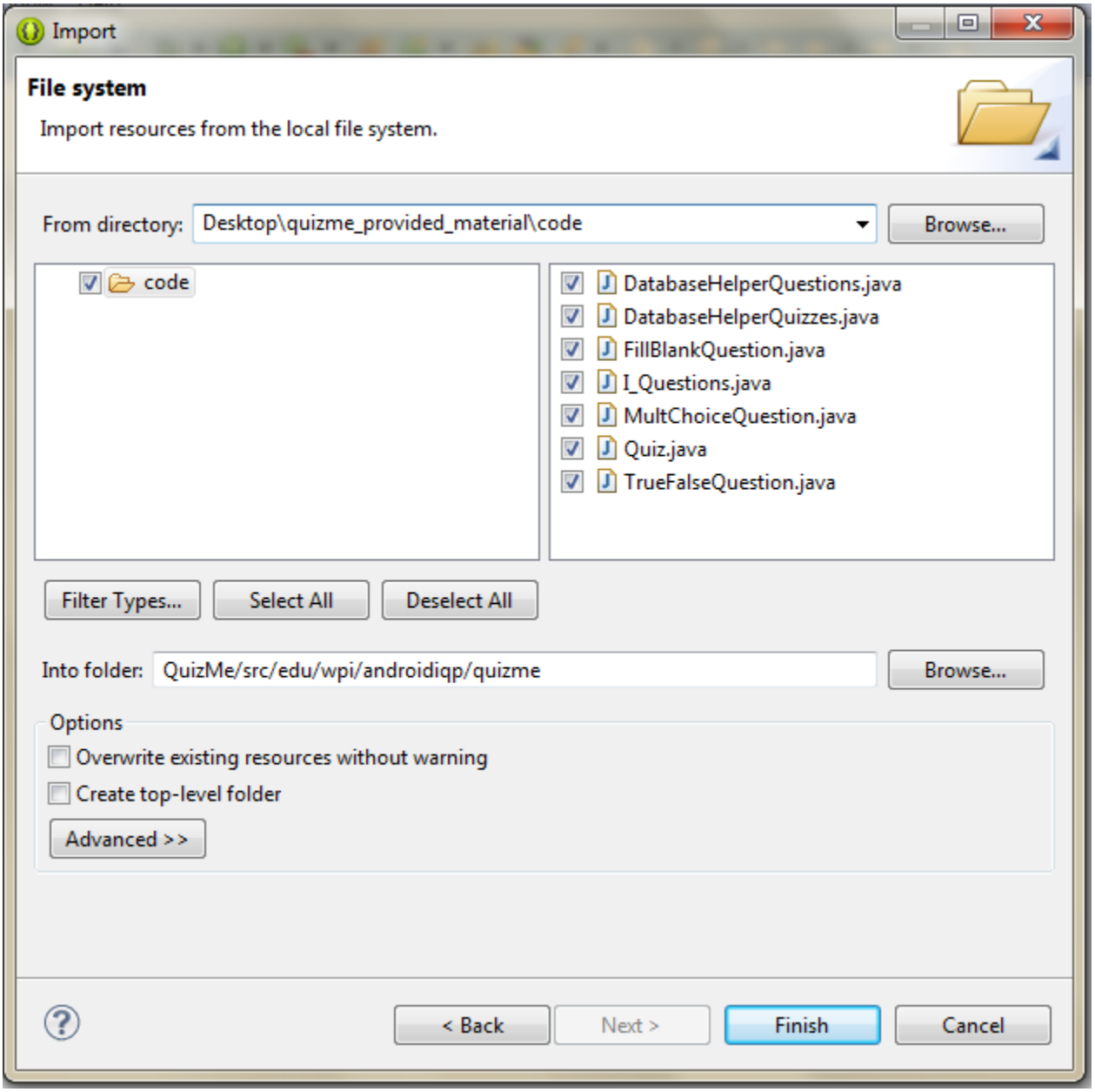

Figure 4.6: Import the provided code into your project

Click **Finish**, and the provided code should be imported successfully.

Next, we will import the various image files that will be used in this project: the application icon, a WPI logo for the splash screen, and a trash can icon. In the Project Explorer panel on the left side of the Eclipse window, navigate to the *res* folder of the QuizMe project and delete the folders *drawable-hdpi*, *drawable-ldpi*, *drawable-mdpi*, and *drawable-xhdpi* (Figure 4.7).

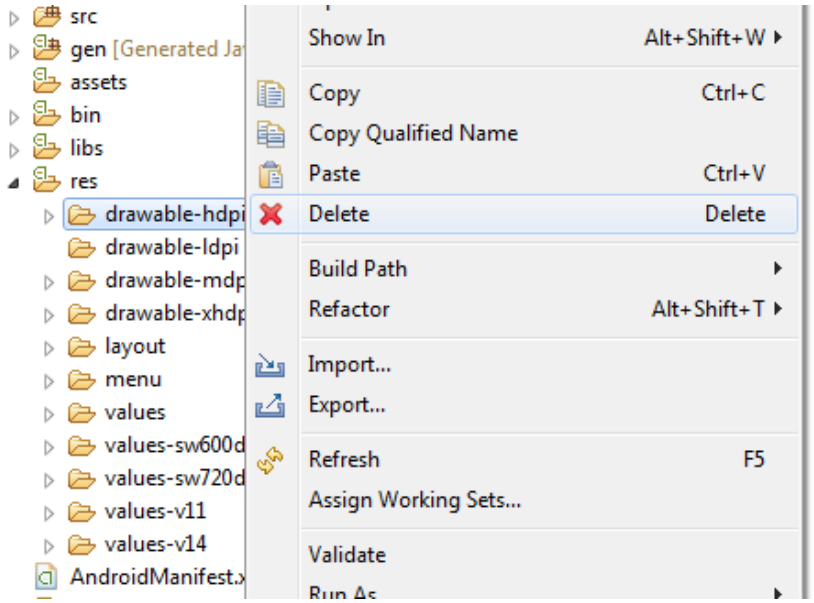

Figure 4.7: Deleting a folder from the project.

Next right-click the *res* folder in the Project Explorer and select **Import**. Similarly to importing the starter code, navigate to the directory in which you extracted the files and select the *Images* folder. In the Import dialog, expand the list of subfolders under *Images* and select each of these subfolders in order to import them into your project. Refer to Figure 4.8.

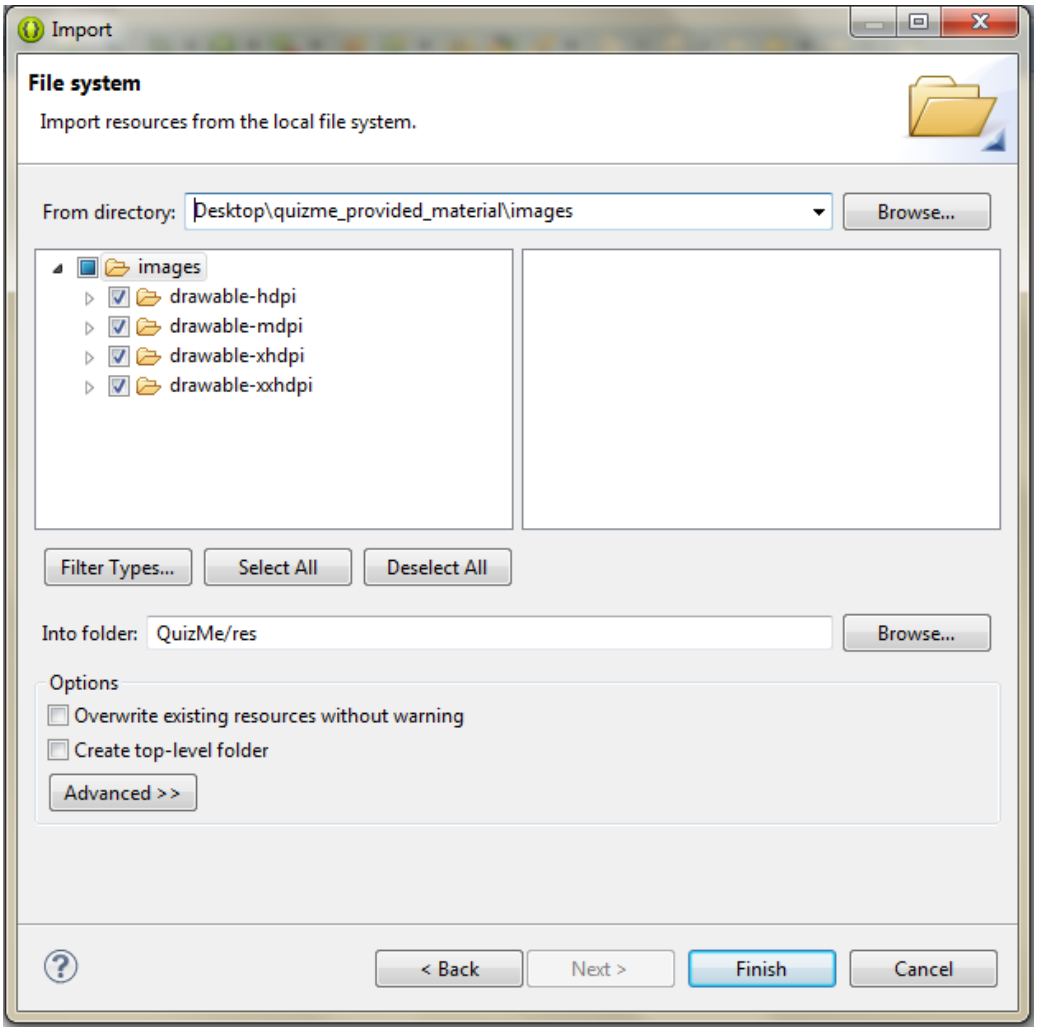

Figure 4.8: Import image folders

Click **Finish**. Next, in the Project Explorer, scroll down to the bottom of the project and find *AndroidManifest.xml.* Double-click this file to open it in Eclipse. In the menu that appears, change the package field to “edu.wpi.it270x.quizme” and click save, and when prompted to change the launch configuration select No (Figure 4.9).

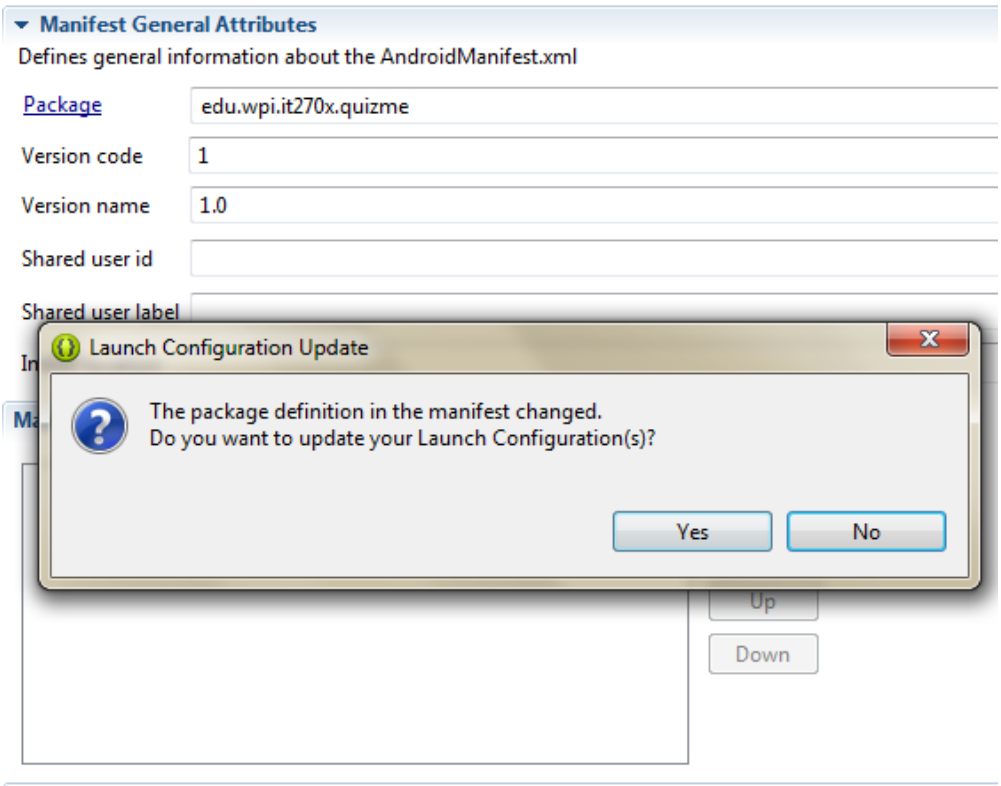

Figure 4.9: Editing the Android Manifest

This change has caused Eclipse to automatically add a line to the *MainActivity.java* file in the src folder. Open that file and expand the list of imports by clicking the "+" in the `import edu.wpi.it270x.stubapp.R;` line. Now change the line from `import edu.wpi.it270x.stubapp.R;` to `import edu.wpi.it270x.quizme.R;` to resolve the error.

Finally, in the Project Explorer, navigate to *res→values* and open *strings.xml*. Change the value of app_name to "QuizMe" by altering the XML code line from

```
<string name="app_name">Stub App</string>
```

to

```
<string name="app_name">QuizMe</string>.
```

You are now ready to start building the QuizMe Application.

4.2: Building the Application: Overview

With the new QuizMe project set up and provided code imported, it is time to start developing the application. QuizMe will store quizzes and their respective Questions in SQL databases, and each Activity in this application will pull the necessary information from these databases. Users will be greeted by a Splash Screen, then prompted to either create a new quiz if none are present or to choose an existing quiz to work with. Quizzes can be run, allowing the user to choose an answer for each Question and receiving instant feedback about whether this

answer is correct. Quizzes can also be edited, allowing the user to add, remove, or edit Questions. Initially, QuizMe will only contain True or False Questions, but additional Question types will be added down the road.

4.3: Splash Screen

Overview

A Splash Screen (Figure 4.10) is not necessary to the app's functionality, but it advertises some key information regarding the application. QuizMe's Splash Screen will display the WPI logo, the application name, and a short description of the application. In general, some application Splash Screens prompt for user interaction before continuing, while others display for a limited amount of time and close themselves; QuizMe utilizes the former.

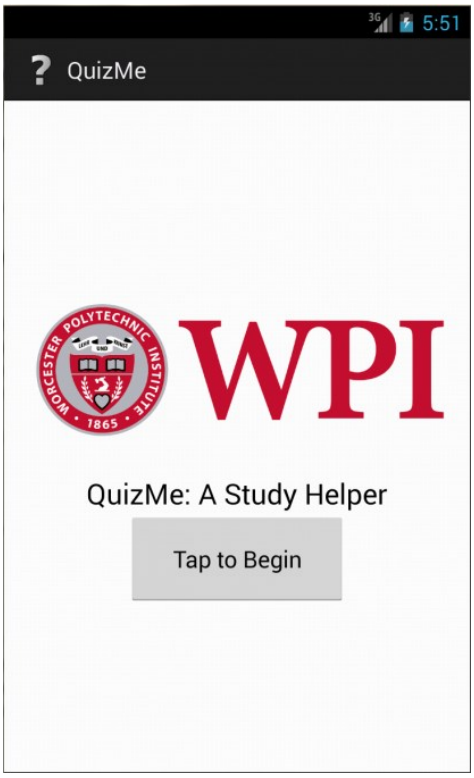

Figure 4.10: The QuizMe Splash Screen

Development

To begin, the XML code controlling what appears on the screen must be altered. XML is a very flexible language in that the only restriction is formatting; the actual fields, which we will be working with shortly, are read in by the program to be utilizing them. For Android development, these XML files consist of graphical layout information.

In the project explorer, expand the QuizMe project, then expand the *res* folder, then *layout*. Double-click *activity_main.xml*. You should be greeted by a graphical representation of the Activity layout. Disregard this for now and click the *activity_main.xml* tab on the bottom of the editor window (Figure 4.11). Working directly with the XML code allows for more precise editing of the layout than the graphical view, though the graphical view does have its uses.

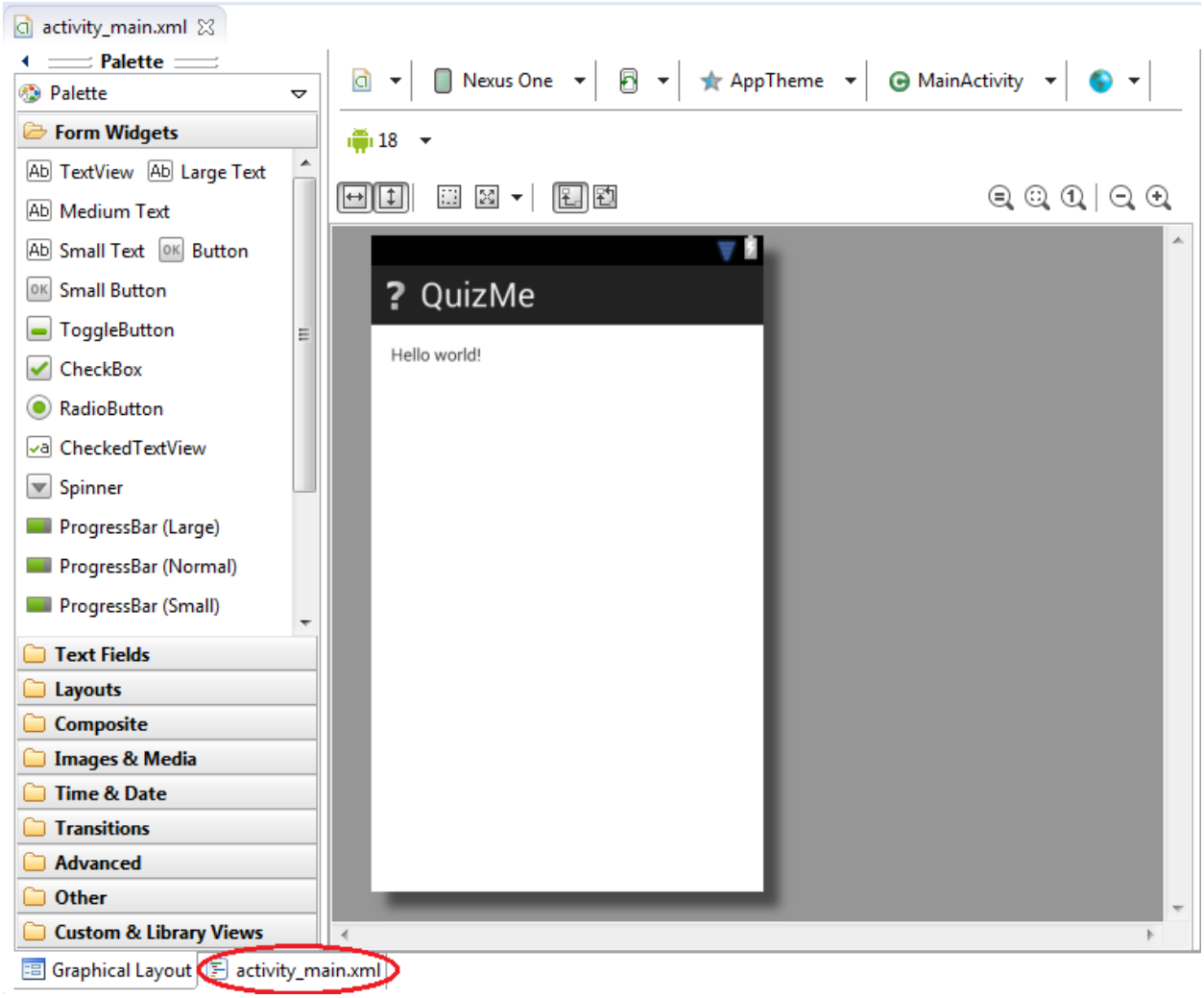

Figure 4.11: The graphical layout tool. Select the circled tab.

Delete all of the XML code in this file and replace it with this:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:tools="http://schemas.android.com/tools"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:gravity="center"
      android:orientation="vertical"
      android:paddingBottom="@dimen/activity_vertical_margin"
      android:paddingLeft="@dimen/activity_horizontal_margin"
      android:paddingRight="@dimen/activity_horizontal_margin"
      android:paddingTop="@dimen/activity_vertical_margin"
      tools:context=".MainActivity" >
```

```
        <ImageView
        android:id="@+id/wpi_logo_splash"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:src="@drawable/wpi_logo"
        android:contentDescription="@string/wpi_logo"/>

        <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/splash"
        android:textAppearance="?android:attr/textAppearanceLarge" />

        <Button
        android:id="@+id/splash_button"
        android:layout_width="170dp"
        android:layout_height="72dp"
        android:text="@string/splash_button" />

</LinearLayout>
```

As you can see, this Activity uses a Linear Layout similar to the one in the stub application. New fields in this one are the `android:padding` field which create a small border of white space around the edges of the layout, and the `tools:context` which designates this layout as the Main Activity of the application. The linear layout contains three objects, an image (`ImageView`), a line of text (`TextView`), and a button (`Button`). The `android:src` field points the application to where the actual image file is stored. The `android:contentDescription` field of `ImageView` provides a textual explanation of the given image.

The next thing to do is to go into *strings.xml* as you did in the Stub App and add the strings “WPI Logo” with the ID “wpi_logo”, “QuizMe: A Study Helper” with the ID “splash”, and “Tap to Begin” with the ID “splash_button”.

Check the graphical layout tab again. Now you should see the completed Splash Screen. Now double-click on the *src* → *edu.wpi.it270x.quizme* → *MainActivity.java* file to open Completely remove the options menu method, as it is not necessary for this Activity:

```
public class MainActivity extends Activity {
    ...

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is
                present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
```

```
}
```

Now replace the onCreate method with the following code (added code is denoted by bold text):

```
import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;

public class MainActivity extends Activity {

    boolean status;
    EditText textbox;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        textbox = (EditText) findViewById(R.id.textbox);
        Button button = (Button) findViewById(R.id.thisbutton);

        status = false;

        button.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View arg0) {
                if(status) {
                    textbox.setVisibility(View.VISIBLE);
                }
                else {
                    textbox.setVisibility(View.INVISIBLE);
                }
                status = !status;
            }
        });

        Button begin_button = (Button) findViewById(R.id.splash_button);

        begin_button.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View arg0) {
                Intent newact = null;
```

```
                startActivity(newact);
            }
        });
    }
}
```

After inputting this code you may see some errors. Go through the code you just input and hover your mouse over every red underlined error. Typically, these errors will be resolved by importing an Android implementation. If you see an option such as the one in Figure 4.12 below, click it. You may also use Ctrl + Shift + O on Windows and Linux (Command + Shift + O on Mac OS X) to find and organize any imports automatically.

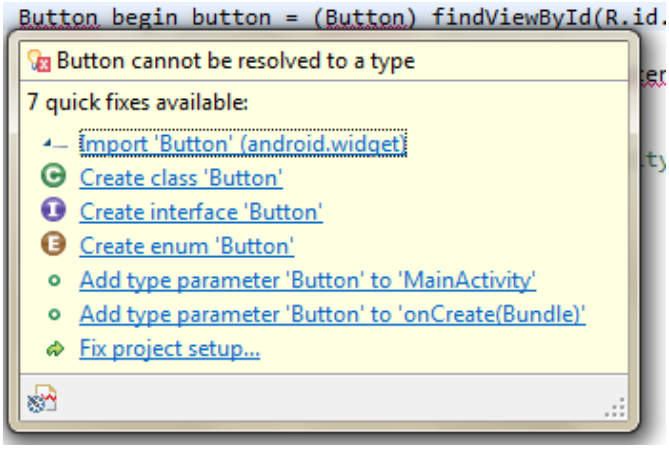

Figure 4.12: The Import option in the error solutions menu

View must be imported the same way. In future development, make sure to check for this if an object is giving you errors. Many errors can simply be resolved by fixing imports. For any other errors, Eclipse will still offer suggestions that may help to resolve the error. By hovering over an error in your code, Eclipse will usually provide information that will be helpful in resolving the error.

At this point, you should have an Android emulator set up already (if not, please refer to Chapter 3 - Stub App). Right-click on the QuizMe project and click **Run As → Android Application**. Wait for the emulator to load (this may take upwards of 10 minutes on a slow system, especially if it is your first time running the emulator). Once the emulator loads, unlock it by sliding the lock orb in any direction, and wait for QuizMe to install and launch. If everything went well, you should see QuizMe startup with the new Splash Screen. Click the button and see what happens.

Well, that did not go so well. Now that QuizMe has crashed, how do you find the source of this problem? Minimize the emulator (do not close it, it will take some time to start up again and we will want to be testing constantly). In Eclipse, in the editor window, there should be some tabs near the bottom or on the right side (Figure 4.13). Click the LogCat tab if it has not already been opened. You should see some red text in this tab. If not, scroll up until you do. Scanning this text will tell you what error the program encountered and at what exact line in the code that it crashed at; in this case, it crashed because the button redirects to a null Activity.

Troubleshooting errors using this console will become commonplace during development. Do not be discouraged if you have to do this often.

Figure 4.13: LogCat may be located the side or bottom of Eclipse

4.4: List of Quizzes

Overview

Now that the Splash Screen is completed, it is time to start working on the meat of the application. We will need to list the available quizzes in order to do anything with this app, and clicking a quiz should open a new Activity, which will be developed later. This list will be created and populated using an Android **ListView** class. A ListView is essentially a clickable list. In the Activity you will create, a ListView is used to store Quizzes pulled from the database. Clicking a ListView entry will bring the user to a new Activity, and newly added quizzes will be added to the list when this Activity is loaded again (Figure 4.14).

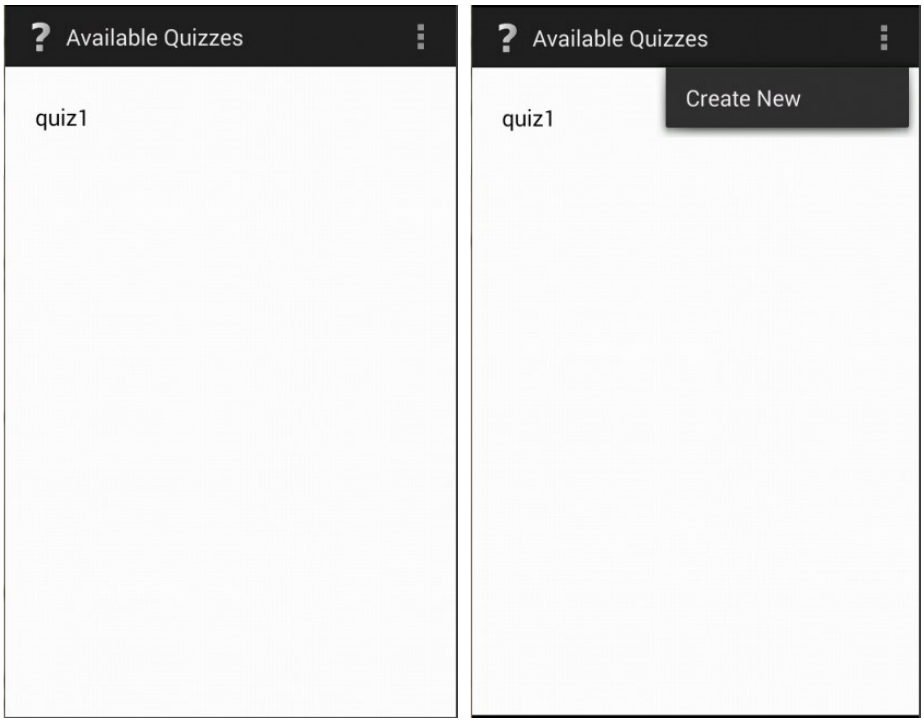

Figure 4.14: QuizListActivity with a single quiz and an options menu

SQLite

The two database helper classes used in QuizMe have been provided in their entirety. However, these should still be studied in detail. A SQL database is, in its most basic form, a collection of tables. Each table has a name, and each record in a table has an automatically generated numerical ID in addition to any other fields required by the database helper class. In this app, *DatabaseHelperQuestions* and *DatabaseHelperQuizzes* contain methods to store, update, or delete Questions and quizzes in databases, as well as receive a list of all of the quizzes or Questions (with a given quiz ID) in the database.

A Note About SQL Injection

SQL injection is the act of manipulating user input so as to corrupt an SQL database. Before database security practices were improved, this was a common way for malicious users to steal users' passwords and other information from important databases. Someone would simply type an SQL command into an input field and they could manipulate the database as they wish.

During development of QuizMe and HomeworkHelper, students with SQL experience may wish to try entering SQL commands, with and without quotes, into any input fields in order to test for SQL injection. While this is not a threat in these applications, as no confidential information should be held in these databases. In fact, as the provided database functions are parameterized (they expect certain results and store these into pre-existing variables), SQL injection should not be possible, and it has been thoroughly tested by the development team.

While QuizMe and HomeworkHelper do not contain sensitive information such as passwords or credit card numbers, we still do not want users to accidentally corrupt their own database by entering quotes or words in SQL commands (what if they want to take a quiz *on* SQL commands?). As such, testing for SQL injection is good practice when working with databases.

Comments

Before we dive into development, we must cover **code comments**. Programming is not only about making a program that runs. Code needs to be readable and future developers (and your future self) need to be able to follow it. This is what code comments are for. In most major programming languages, Java included, comments are created like so:

```
// This is a comment

/* Everything between
   these two symbols
   is a comment
*/
```

A comment should describe what is going on in the code. Do not try to explain every last detail of your code in comments, however. Explain why your program is doing what it is doing, with just some explanation as to how. Comments should be written during development, not after (though adding comments afterward or cleaning up what you already have is never a bad thing, and we will ask you to do that at the end of this tutorial).

This project guide will show very little comments in provided code, with explanations in the surrounding paragraphs. You will be expected to create comments for both provided code as and for any code that you write yourself.

Development

First a new Activity must be created. Right-click on your project in the Project Explorer and select **New→Other**. Then drop down the Android folder and choose **Android Activity**. Click **Next**. On the following screen choose **Blank Activity** and press **Next** again. Name the Activity "QuizListActivity" (Figure 4.15) then click **Finish**.

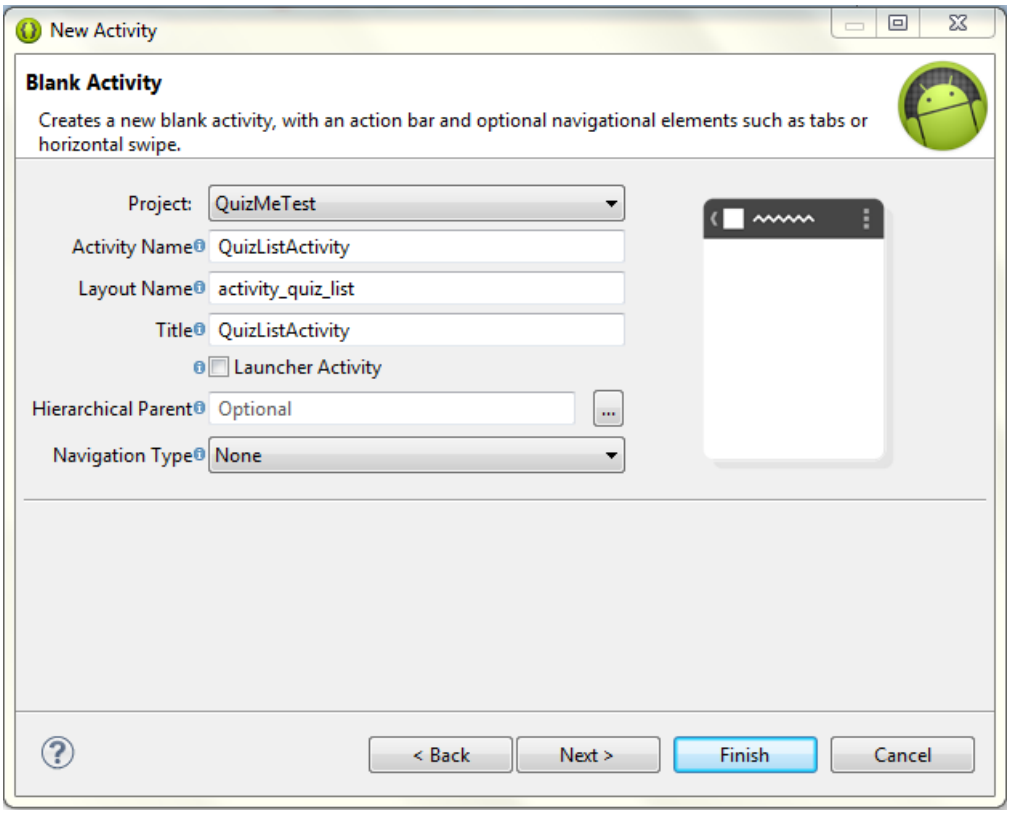

Figure 4.15: Creating QuizListActivity

This alters and creates some files but the important additions are *QuizListActivity.java* and *activity_quiz_list.xml*. The *activity_quiz_list.xml* file should automatically open in graphical mode after you click finish (if not, open it) and click the tab at the bottom to view the XML code. Replace the TextView object with a Listview and you will be done with the XML for this Activity:

```
<RelativeLayout xmlns:Android="http://schemas.android.com/apk/res/android"
```

```
        xmlns:tools="http://schemas.Android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        tools:context=".QuizListActivity" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/hello_world" />

        <ListView
            android:id="@+id/mainlist"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" >

        </ListView>
</RelativeLayout>
```

Next open the *QuizListActivity.java* file and add fields for both database helper classes, we will need them later.

```
public class QuizListActivity extends Activity {

    DatabaseHelperQuizzes db;
    DatabaseHelperQuestions qdb;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
      …
```

Leave the onCreate method as it is. Instead we will be doing our programming inside a method called onStart. To understand why let us first look at how an Activity behaves through several of its methods. This is what we call the Android Activity lifecycle:

onCreate: Code written in this method only runs once, when the Activity is first created.
onDestroy: Code written in this method runs when the Activity closes.
onStart: Code written in this method runs whenever the Activity becomes visible on the screen.
onStop: Code here runs when the Activity is no longer visible on the screen.

When another Activity is started (or in Android terminology, when an Activity is **inflated**) the Activity that started it is still running in the background. Because this Activity displays a list of available Quizzes, it should be refreshed every time it becomes visible in order to refresh the list. As such, most of the code for this Activity will be written in the onStart method. The first

thing you will want to do is override the onStart method, after which you will fill in the database fields with actual objects. This is done because onStart technically exists in one of your imported files already, so we override that one to use ours. Next, you will retrieve a list of from the database you defined using a provided method called getAllQuizzes which returns all Quizzes stored in the database in the form of a list.

Edit *QuizListActivity* as follows:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
	super.onCreate(savedInstanceState);
	setContentView(R.layout.activity_quiz_list);
}

@Override
protected void onStart() {
	super.onStart();

	// Open database
	db = new DatabaseHelperQuizzes(this);
	qdb = new DatabaseHelperQuestions(this);

	//Get list of quizzes
	LinkedList<Quiz> available_quizzes = db.getAllQuizzes();
```

Next we will define a ListView object the same way a Button was defined in the Stub App. Once this is completed, we will convert the list of Quizzes to an **array** and use a construct called an **Array Adapter** to bind the array of Quizzes to the ListView.

```
//Get list of quizzes and convert to array
LinkedList<Quiz> available_quizzes = db.getAllQuizzes();
ListView quiz_list = (ListView) findViewById(R.id.mainlist);

int i = 0;
Quiz arrayOfQuizzes[] = new Quiz[available_quizzes.size()];
for(Quiz tmp1: available_quizzes){
	arrayOfQuizzes[i] = tmp1;
	i++;
}

ArrayAdapter<Quiz> adapter = new ArrayAdapter<Quiz>(this,
		Android.R.layout.simple_list_item_1, arrayOfQuizzes);
quiz_list.setAdapter(adapter);
```

This is probably the simplest way to display items in a list. The ArrayAdapter constructor takes three arguments, the context (`this`), a display format (`Android.R.layout.simple_list_item_1`), and the array to be displayed. Of these three the one that needs explanation is the format. The format field tells the Android system what to

display on each item of the list and a format of `simple_list_item_1` informs the system that the toString method of each item will determine what displays during runtime.

Arrays may seem complicated, but they are very straightforward. An array is essentially a list, and each list entry has an index. An array is declared in Java by appending brackets [] to the variable name. In the initial declaration, anything inside the brackets defines the size of the array. In later calls to the array, the number in the brackets will represent the index. The **for** loop above is a basic method of iterating through an array and working on each value. We use an integer **i** to store the current index. The first line of the loop says that for each iteration through the loop, set tmp1 equal to the current index of available_quizzes. Then, set arrayOfQuizzes at index i equal to tmp1, and increase i by 1. This way, we keep moving through the array, setting each value appropriately until we reach the end of the available_quizzes array.

Before proceeding further, a better toString method should be defined in the *Quiz.java* file.

```
public class Quiz {

    private int _id;
    private String name;

    …

    public int get_id() {
       return this._id;
    }

    @Override
    public String toString() {
       return this.name;
    }
}
```

This is more useful than the native toString method because this method only returns the quiz name whereas the native method would return the name and _id fields of a quiz as an appended string. We never need the ID number as a string in this application, just the quiz name. Now go back to the *QuizListActivity.java* file.

What happens when an item is clicked must be handled using the ListView's version of an onClickListener called onItemClickListener, as defined below:

```
        quiz_list.setAdapter(adapter);

        //create a listener for clicks on the list
        quiz_list.setOnItemClickListener(new OnItemClickListener() {

              @Override
              public void onItemClick(AdapterView<?> parent, View v, int
                                      position, long id) {
```

```
                Quiz clicked = (Quiz) parent.getItemAtPosition(position);
                // stuff will happen here later
            }

        });
}
```

You have now finished the onCreate method for this Activity. Next is a short onStop method that will close the database fields. The overall result is that when the Activity becomes visible, the database objects are created and when the user switches to a new Activity, the databases are closed and no longer accessible.

```
            public void onItemClick(AdapterView<?> parent, View v, int
        position, long id) {
                Quiz clicked = (Quiz) parent.getItemAtPosition(position);
                //stuff will happen here later
            }
        });
}

// Close the databases
@Override
protected void onStop() {
        super.onStop();
        try{
            db.close();
            qdb.close();
        }finally{
            //do nothing
        }
}
```

We wrap these close statements in a try{}finally block for error handling. The program tries to close the databases, but if it cannot, then the try{}finally code avoids crashing the app.

Now an options menu will be added with the option to create a new quiz. When a new Activity is created using Eclipse, an options menu is automatically generated. Menu files are stored in the *res* → *menu* folder and the menu associated with this Activity is *quiz_list.xml*. You should be able to infer this from the createOptionsMenu method that has been automatically created. Open *quiz_list.xml* and alter the code as follows:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
        <item
            android:id="@+id/action_settings"
            android:orderInCategory="100"
            android:showAsAction="never"
            android:title="@string/action_settings"/>

        <item
            android:id="@+id/action_create_quiz"
```

```
            android:orderInCategory="100"
            android:showAsAction="never"
            android:title="@string/action_create_quiz"/>
</menu>
```

Now add a string to the *strings.xml* file prompting the creation of a quiz and give it the ID "action_create_quiz".

In order to perform an action based on what is selected in the options menu, another method must be added to *QuizListActivity*, onOptionsItemSelected. Like similarly named methods, this method determines what happens when an options menu item is tapped.

```
public boolean onCreateOptionsMenu(Menu menu) {
      // Inflate the menu; this adds items to the action bar if it is
present.
      getMenuInflater().inflate(R.menu.quiz_list, menu);
      return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
      // Handle item selection
      switch (item.getItemId()) {

            // Create Quiz option starts a new activity
            case R.id.action_create_quiz:
                  //CHANGES WILL BE MADE HERE IN FUTURE
                  return true; // Return so we don't fall through cases

            default:
                  return super.onOptionsItemSelected(item);
      }
}
```

As you can see the method uses a switch statement along with the IDs defined in the menu XML file to selections. A switch statement takes in a value and checks this value against varous cases. If it matches any case, the code for that case will be executed, as well as code for any case below this case in the code. That is why we **return**, or exit the method and give a value, at this point.

The very last thing to do now is inflate this Activity from the Splash Screen Activity created a while back. Go to your *MainActivity.java* file and make the following alterations.

```
      begin_button.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View arg0) {
                  Intent newact = null;
                  Intent newact = new Intent(MainActivity.this,
                                          QuizListActivity.class);
```

```
                startActivity(newact);
            }
        });
```

An Intent tells what Activity will be launching from another Activity. In this case, *QuizListActivity* is launching from *MainActivity*. Passing this Intent through the built-in startActivity method does just what you would expect, it starts the Activity from the Intent.

Now when you test your code in the emulator, pressing the button should start *QuizListActivity* and display an empty list.

4.5: Creating a Quiz

Overview

The ListView appears to be complete, but how can we know for sure without any Quizzes in the list? Users will need to be able to add quizzes to the database (Figure 4.16). In order to do this, a new Activity should be created, accessible through the **options menu** of *QuizListActivity.*

*It would be wise to copy and modify code from the Stub App for this new Activity, as it utilizes a similar layout. This section will mostly be left for you to develop on your own.

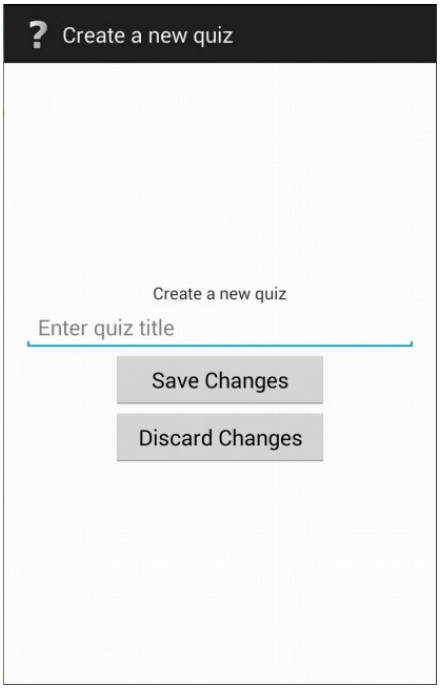

Figure 4.16: AddQuizActivity

Development

Create a new Activity called “AddQuizActivity”. Now in *AddQuizActivity.java,* render the options menu unavailable as we will not require it here.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
```

```
        // Inflate the menu; this adds items to the action bar if it is
present.
        getMenuInflater().inflate(R.menu.add_quiz, menu);
        return true;
}
```

Now it is time to test what you've learned by doing some work without much guidance. First, alter the *activity_add_quiz.xml* file so that it can support the creation of Quizzes. This will require an EditText to hold the name of the Quiz, as well as a save button and a discard button.

Next alter the *AddQuizActivity.java* file. This Activity will either exit when the discard button is pressed, or save the quiz in the quiz database when the save button is pressed. Some provided methods you will need to utilize are listed below:

`EditText``.getText().toString();`
This method takes whatever is typed inside an EditText and returns it as a string.

`DatabaseHelperQuizzes``.addQuiz(Quiz q);`
This method adds Quiz q to the database if a Quiz of that exact name is not already in the database. The _id field of the Quiz object should be blank as it is filled in automatically through SQL.

`finish();`
This method causes the currently running Activity to exit.

Testing the Application

The application should be tested before continuing further. If the Quiz database is not functioning correctly, it is impossible to go on with this project. Run the QuizMe project on the Android emulator.

If at any point the application crashes, check the console and find the red error text. Scanning this text should tell you what type of error was encountered, which file the error was encountered in, and the line number where the program crashed.

Once the application is loaded, continue past the Splash Screen. Initially, the database is empty, and the list will be blank. Create a new Quiz with any name. You should automatically return to *QuizListActivity.java* on pressing your **save** or **discard** buttons. Open the options menu and create another new Quiz with the same name as the previous one. You should not be able to, as the database method disregards the operation if the name is found already inside the database. Change the name and save changes, and a new Quiz should be added to the list. Start creating a new quiz one more time, but this time, discard changes. Nothing should have changed. If any of these scenarios did not occur correctly for your app, you must troubleshoot these issues. Step through your code mentally (or on paper) and see if you can track down the issue. Think it through, why would your code not be doing what it should be doing when a particular action occurs?

Once you have sorted out any bugs, continue to the next section.

4.6: Quiz Options

Overview

What good is a Quiz if it has no Questions? How will we go about running a Quiz? To solve these problems, we will add a *QuizOptionsActivity* (Figure 4.17). This Activity will provide a basic user interface to Run or Edit the selected Quiz, with an option to delete the Quiz (the Delete option is placed in the options menu to prevent accidental deletion of Quizzes). We will have to edit *QuizListActivity* to open this Activity and pass through the database ID number of the selected quiz. The Run Quiz button will not do anything just yet, but we will be working with the Edit Quiz button shortly.

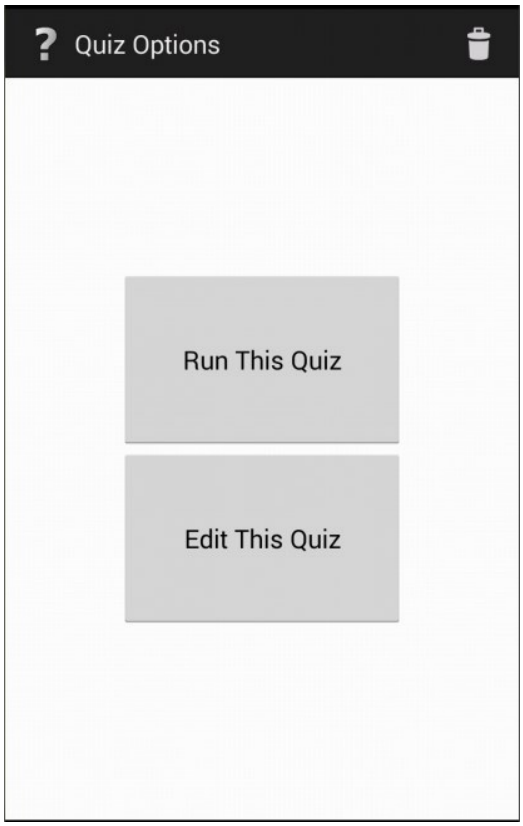

Figure 4.17: QuizOptionsActivity

Development

Create a new Activity called “QuizOptionsActivity”. In order to work with a Quiz, the Quiz’s SQL ID must be passed from the Activity that inflates QuizOptionsActivity. We can pass Extras through Intents in order to pass data through activities. To do so, make the following changes to *QuizOptionsActivity.java*:

```
public class QuizOptionsActivity extends Activity {

    public static final String QUIZ_ID_PASSED =
                               "edu.wpi.it270x.quizme.passednum";
    private int quiz_id;
```

```
DatabaseHelperQuizzes db;
DatabaseHelperQuestions qdb;

@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_quiz_options);
}

@Override
protected void onStart() {
   super.onStart();

   // Create the databases and grab info from previous Activity through
            Intent Extras
   db = new DatabaseHelperQuizzes(this);
   qdb = new DatabaseHelperQuestions(this);
   quiz_id = getIntent().getIntExtra(QUIZ_ID_PASSED, -1);

}
```

Data passed between activities uses a String key to be recognized, in this case the String key is the variable `QUIZ_ID_PASSED` which is a static final String to ensure it does not change (**static** guarantees that this key will not be created for every instance of this Activity, only one copy of this key will exist. **final** states that this value cannot be changed). The getIntent method retrieves information about the currently running Activity, which then has different methods such as `getIntExtra(String key, default)` which searches for an Intent extra with the specific key and uses the default value if no extra with that key is found. Now that this Activity searches for an extra, the code in *QuizListActivity.java* must be modified to attach that extra.

```
public class QuizListActivity extends Activity {
   …
      quiz_list.setOnItemClickListener(new OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent, View v, int
                                    position, long id) {
                  Quiz clicked = (Quiz) parent.getItemAtPosition(position);
                  // stuff will happen here later
                  Intent newact = new Intent(QuizListActivity.this,
                                             QuizOptionsActivity.class);
                  newact.putExtra(QuizOptionsActivity.QUIZ_ID_PASSED,
                                  clicked.get_id());
                  startActivity(newact);
            }
      });
```

…

As you can see, there is a third step in Activity inflation here. After the Intent object `newact` is created, but before it is inflated, the Quiz's ID (from through the Quiz.get_id() function) is attached to the string defined in *QuizOptionsActivity.java*.

Now back in *activity_quiz_options.xml,* define two buttons, one to edit a Quiz and one to run the Quiz. Create onClickListener methods for these two buttons inside *QuizOptionsActivity.java*. These buttons do not have to do anything yet; we will edit the onClickListener methods soon.

Finally you will be walked through creating an option to delete Quizzes. Make the following changes in *quiz_options.xml* file in the menu folder.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
      <item
            android:id="@+id/action_settings"
            android:orderInCategory="100"
            android:showAsAction="never"
            android:title="@string/action_settings"/>

      <item android:id="@+id/action_delete_quiz"
            android:icon="@drawable/ic_action_discard"
            android:title="@string/action_delete_quiz"
            android:showAsAction="ifRoom" />
</menu>
```

There are several new fields here. The most important is **`android:icon="@drawable/ic_action_discard".`** This field causes an icon to display for this object, found in any of the *drawable-XXXX* folders. These folders contain any images used within an application. Devices with different resolutions will pull icons from the respective folder for that resolution.

Now add a string with the ID "actions_delete_quiz" to your *strings.xml* file. Finally, go back to *QuizOptionsActivity.java* to make changes to the menu for deleting a Quiz.

```
public boolean onCreateOptionsMenu(Menu menu) {
      // Inflate the menu; this adds items to the action bar if it is
present.
      getMenuInflater().inflate(R.menu.quiz_options, menu);
      return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
       // Handle item selection
       switch (item.getItemId()) {

       // Delete quiz
       case R.id.action_delete_quiz:
```

```
            AlertDialog.Builder alertDialogBuilder2 = new
AlertDialog.Builder(this);
            alertDialogBuilder2.setTitle("Delete");
            alertDialogBuilder2.setMessage("Are you sure you want to delete
this quiz?");
            alertDialogBuilder2.setCancelable(false); // Sets whether this
dialog is cancelable with the BACK key.
            // "Yes, exit the app"
            alertDialogBuilder2.setPositiveButton("Yes",new
DialogInterface.OnClickListener() {
                  public void onClick(DialogInterface dialog,int id) {
                        qdb.clearQuiz(quiz_id);
                        db.deleteQuiz(new Quiz(quiz_id, ""));
                        finish();
                  }
            });
            // "No, I changed my mind"
            alertDialogBuilder2.setNegativeButton("No",new
DialogInterface.OnClickListener() {
                  public void onClick(DialogInterface dialog,int id) {
                        dialog.cancel();
                  }
            });
            AlertDialog alertDialog2 = alertDialogBuilder2.create();
            alertDialog2.show();
            return true;

      default:
            return super.onOptionsItemSelected(item);
      }
}
```

As you can see there is a lot of code in this menu under the case of "action_delete_quiz". On closer inspection, however, most of it is actually fairly simple, defining text and buttons for a confirmation before running a single database method. This utilizes the switch statement seen earlier, with just one case. If the delete icon is tapped, an **AlertDialog** is created - a popup, essentially. This dialog is not cancelable with the Back button and has two options. Pressing Yes will first delete the Questions from the Question database with the appropriate quiz ID. Next, the quiz itself will be deleted. Finally, the Activity will close. The dialog is canceled if the user chooses No, and nothing else happens.

Testing

Load the application into the emulator again and test this Activity. Press both buttons, and try to delete the current quiz. When testing an Android Activity, you want to test every feature as well; do what you can to try to cause the application to crash or for something to mess up, such as the database. In this case, ensure that deleting the quiz removes it from the database, which you can see on the previous Activity. Try whatever you can think of to break

the application. If you find anything that does not work properly, go back into your code and see what you can do to fix it.

The "Back" Button

The back button on Android devices (which may be a physical button, a soft touch button, or a button presented by the operating system) has very predictable functionality: bringing the user back one step. By default, during Android development, an application will remember which Activity a new Activity was called from (for example, *QuizOptionsActivity* is called from *QuizListActivity*). Pressing the back button will bring the user back to this previous Activity (running anything that was defined in onStart again but not onCreate). It is possible to manipulate what this button does within an app, but we will not be doing so in this project. If you wish to study this, the Android Developers documentation can be found at the following web page:

http://developer.Android.com/design/patterns/navigation.html

4.7: A List of Questions

Overview

Now that we have two buttons on *QuizOptionsActivity*, we need something for these buttons to do. Each of these buttons leads to a different branch of the app, both of which are connected by the Question database. We cannot run the Quiz without Questions, so we will start with the Edit branch. This new Activity (Figure 4.18) will be called *QuestionListActivity*.

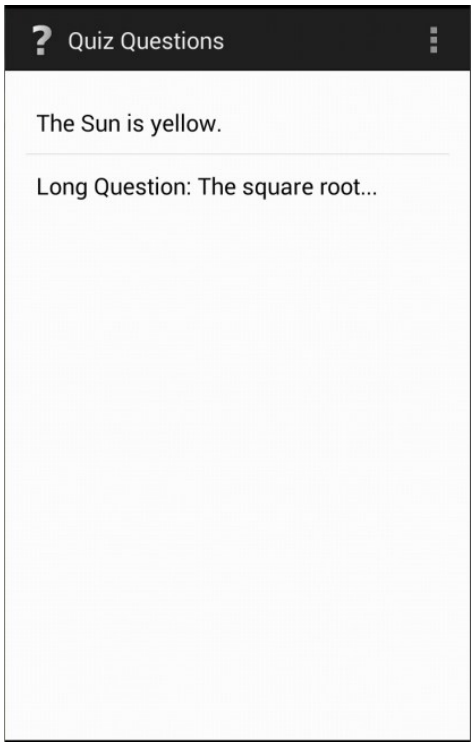

Figure 4.18: QuestionListActivity

Recycling Code

When the Edit Quiz button is tapped, a list of Questions in the appropriate Quiz should be loaded. But wait, we already have a ListView in a previous Activity, and this ListView interacts with the database in the same way that we need *QuestionListActivity* to! We can simply copy and modify the code from *QuizListActivity* to suit this new purpose.

Code recycling may seem like an easy way out, but a programmer's time is valuable. If you have already written similar code once before, why write it again? By reusing and modifying code you have previously written, you not only speed up the development process, you end up writing more consistent code throughout a project as well as establishing your own coding style. Just be sure to modify the code as need be, and that you are not creating even more work for yourself by recycling code than you would have if you started from scratch.

Development

You will actually be developing this Activity mostly on your own, as you should have developed the necessary skills by now from previous activities.

Create a new Activity called "QuestionListActivity". Modify the XML file for this Activity to display a ListView. Next, modify the **Edit Button** onClick method you created in *QuizOptionsActivity.java* and make it inflate *QuestionListActivity* and pass the Quiz ID through as an extra. Now we can get to this new Activity.

Begin recycling code from *QuizListActivity.java*. Changes will need to be made of course. The LinkedList used in this Activity must be changed to a LinkedList of <IQuestions>, and the function that returns the list will be `DatabaseHelperQuestions.getQuizQuestions(int quizID)` which returns a linked list of all Questions associated with the given Quiz ID. Populating the list using an Array Adapter will be extremely similar; you should not have to modify that code much. Because the Array Adapter is using a toString method, be sure to add this line to *IQuestions.java*:

```
public String toString();
```

Until now, we have not heavily utilized the *IQuestions* interface. A Java **interface** can be thought of as a template. Any classes that use this interface must use the same methods as the interface, but they can also have their own in addition to those required. For example, think about the final QuizMe product. There are three types of Questions, but each Question has similar fields and methods. In addition, each type has its own methods that define that Question type. A method elsewhere can work on all types of Questions by acting on the interface rather than a particular class, such as the ListView in this Activity. Let's return to the code now.

Next, define the toString methods in *TrueFalseQuestion.java*, *MultChoiceQuestion.java*, and *FillBlankQuestion.java*. The toString method should return the `questText` field, up to 30 characters. If the Question is longer than 30 characters, append it with an ellipses ("...") (hint: in Eclipse, if you begin typing a period after a variable name, Eclipse will list the methods that can be used by that variable's class or type. Use this to find string functions to trim the displayed text and append the ellipses). Finally, the onItemClick method should retrieve the IQuestion that was

clicked; we will modify it shortly. Leave the options menu untouched for now, it will be changed later.

4.8: True or False Questions

Overview

We now have a list to store Questions from the database. In order to for this to be remotely useful, we need to be able to add Questions to the database (Figure 4.19). This process is going to be extremely similar to adding quizzes, except we will need a different user interface for this Activity and it will need to save more values in the database than just a name. This new UI will need allow for Question text and an answer.

This guide will help with development of the Activity itself, but with the skills you have developed so far, you should be able to create an option that launches this Activity without help. If need be, refer back to section 4.2.2.

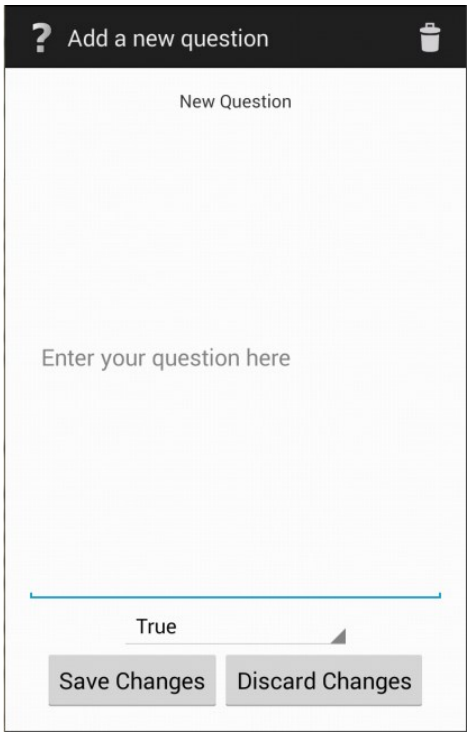

Figure 4.19: AddQuestionTrueFalseActivity

Development

Create a new Activity to create and modify True or False Questions, with the name **"**AddQuestionTrueFalseActivity**"**. You may notice that each type of Question has different fields, yet are all three types displayed in the same list as a result of the interface. As such, deciding which Activity to inflate from *QuestionListActivity.java* can be difficult. Instead, the Question will be made responsible for pointing to its own editor Activity. To do this, open the interface *IQuestions.java* and add the following method:

```
public Intent prepare_Editor(Context c, boolean isNew);
```

Now, define this method in all three Question classes, which will allow the code to compile. In the *FillBlankQuestion.java* and *MultChoiceQuestions.java* files the function will return null; we will modify this later. Inside *TrueFalseQuestion.java,* the method will be defined as such:

```
public class TrueFalseQuestion implements I_Questions {

    private int _id;
    …
    @Override
    public Intent prepare_Editor(Context c, boolean isNew) {

        Intent newact = new Intent(c, AddQuestionTrueFalseActivity.class);

        if(!isNew) {
              /*
              Attach the _id, quizID, questText, and answer
              fields as extras here
              */
        }

        // attach the boolean "isNew" as an extra here

        return newact;

    }
}
```

Follow the instructions in the code comments to finish this method. The isNew boolean value is used to determine if we are creating a new Question or if we are editing an old Question. As such, if we are editing an existing Question, we want to receive the Question's ID, quizID, text, and answer as extras so that we can populate the fields accordingly.

Now go back to *QuestionListActivity.java*. Inside the onItemClick method, call `prepare_Editor(QuestionListActivity.this, false)` on the object clicked in order to get the Intent that will be inflated. Next, alter the options menu in *QuestionListActivity.java* to display the option to create a True or False Question. When clicked, this method perform the following:

```
Intent newactTF = new TrueFalseQuestion(0, "",
"").prepare_Editor(this, true);
startActivity(newactTF);
```

This will cause the menu option to open the proper Question editor Activity. Now that *AddQuestionTrueFalseActivity* inflates properly, it needs to actually do something. This Activity should allow the user to edit the Question's text or answer and based on whether the Question is new or not (determined by the isNew boolean flag). The Activity should use the method

`DatabaseHelperQuestions.``updateQuestion` if the user is editing an existing Question, or **`DatabaseHelperQuestions.`**`addQuestion` to add a new Question to the database. As well, this Activity should have an option to delete the Question. Create a deletion icon in the options menu and use the method **`DatabaseHelperQuestions.`**`deleteQuestion` to delete this Question from the database if this icon is selected. Be sure to create an AlertDialog for confirming deletion as done previously.

If the boolean isNew is False, you must also populate the display with existing information from the passed extras. EditText fields and Spinners both have methods to do this, which can be found in the Android Developers documentation. (We could easily give you the methods here, but research and understanding documentation are necessary skills for software development).

4.9: Running a Quiz

Overview

With all of our Quiz and Question creation activities and methods completed, now we can create one final Activity to run a Quiz (Figure 4.20). We will be implementing **fragments** in this Activity, though they will not be heavily utilized until the first major expansion to this project is completed. Fragments allow for different methods and XML layouts to appear within a frame of an Activity. This will allow us to display the proper fields and run the methods necessary for all three types of Questions in the final product. While we will not be making much use of them until later, implementing fragments now will save us time and cause less headaches down the road. As usual, you can refer to the Android Developers documentation for more information:

http://developer.Android.com/guide/components/fragments.html

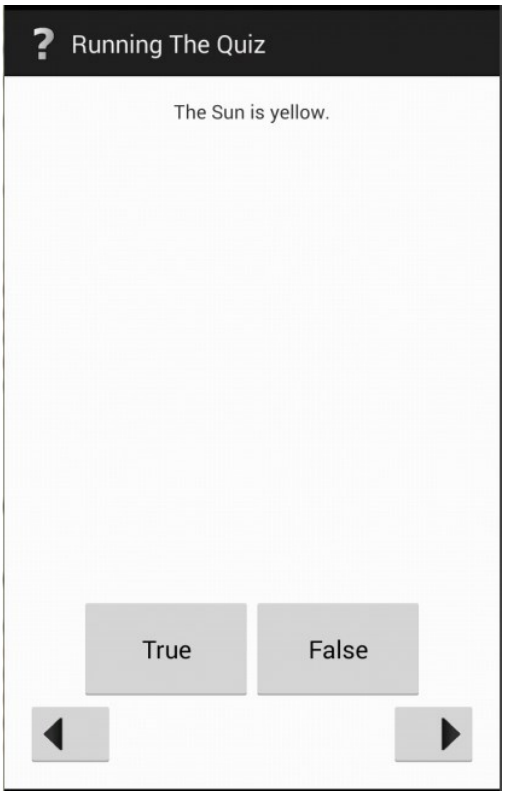

Figure 4.20: RunQuizActivity

Development

Create a new Activity called “RunQuizActivity”. Inflate this Activity when the Run Quiz button in *QuizOptionsActivity.java* is clicked. Be sure to pass the Quiz ID as an extra when inflating RunQuizActivity, as we will need to know which Quiz to pull Questions from. Once the button is complete, open *activity_run_quiz.xml*. This is where the implementation of fragments will begin, specifically allocating a place in the layout for a fragment to display. Replace all the code in *activity_run_quiz.xml* with the following:

```
<LinearLayout xmlns:Android="http://schemas.android.com/apk/res/android"
      xmlns:tools="http://schemas.Android.com/tools"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:gravity="bottom|center"
      android:orientation="vertical"
      android:paddingBottom="@dimen/activity_vertical_margin"
      android:paddingLeft="@dimen/activity_horizontal_margin"
      android:paddingRight="@dimen/activity_horizontal_margin"
      android:paddingTop="@dimen/activity_vertical_margin"
      tools:context=".RunQuizActivity" >

      <FrameLayout
            android:id="@+id/fragmentContainer"
            android:layout_width="match_parent"
```

```
            android:layout_height="wrap_content"
            android:layout_weight="0.50"
            android:gravity="center" />

        <RelativeLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_gravity="bottom"
            android:gravity="bottom"
            android:orientation="horizontal" >

            <Button
                android:id="@+id/next_button"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_alignParentRight="true"
                android:drawableRight="@drawable/arrow_right"
                android:gravity="right|bottom" />

            <Button
                android:id="@+id/back_button"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_alignParentLeft="true"
                android:drawableLeft="@drawable/arrow_left"
                android:gravity="left|bottom" />
        </RelativeLayout>

</LinearLayout>
```

Most of these fields are familiar. However, the Frame Layout is something new. The Frame Layout object here is reserving a space in the display for the fragment to appear. Now before we program *RunQuizActivity.java,* we must create the fragment it will host. Right-click on the package inside the *src* folder and select **new→Class**. Name this class “TrueFalseFragment” (Figure 4.21). Do not finish creating the class just yet, however; in the superclass field, select **Browse**, then type in fragment and select the one that begins with **android.support.v4** (Figure 4.22).

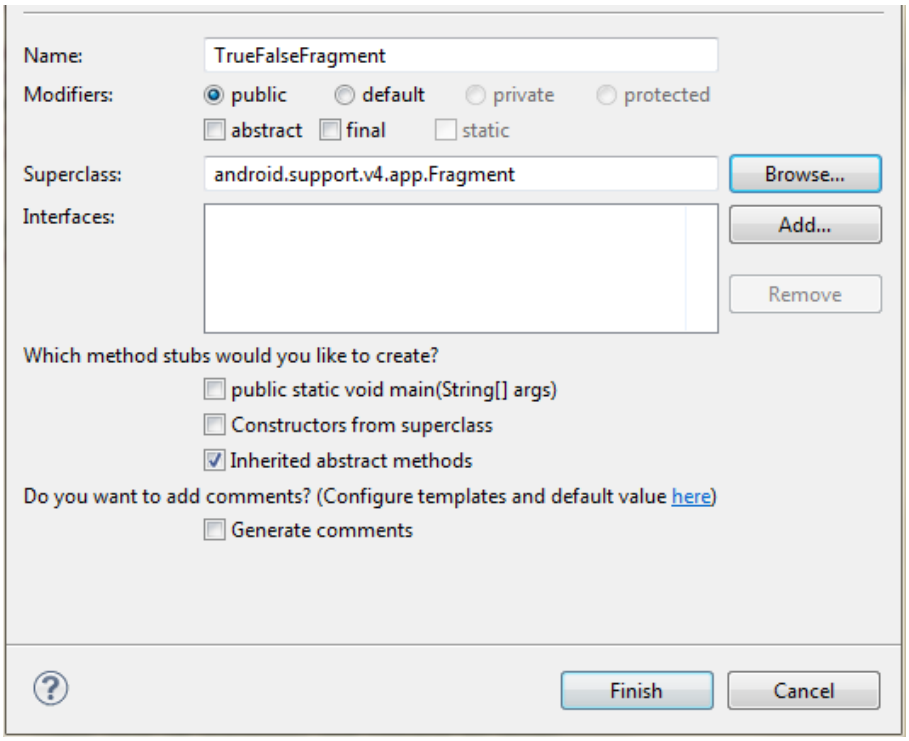

Figure 4.21: Class creation settings for fragments

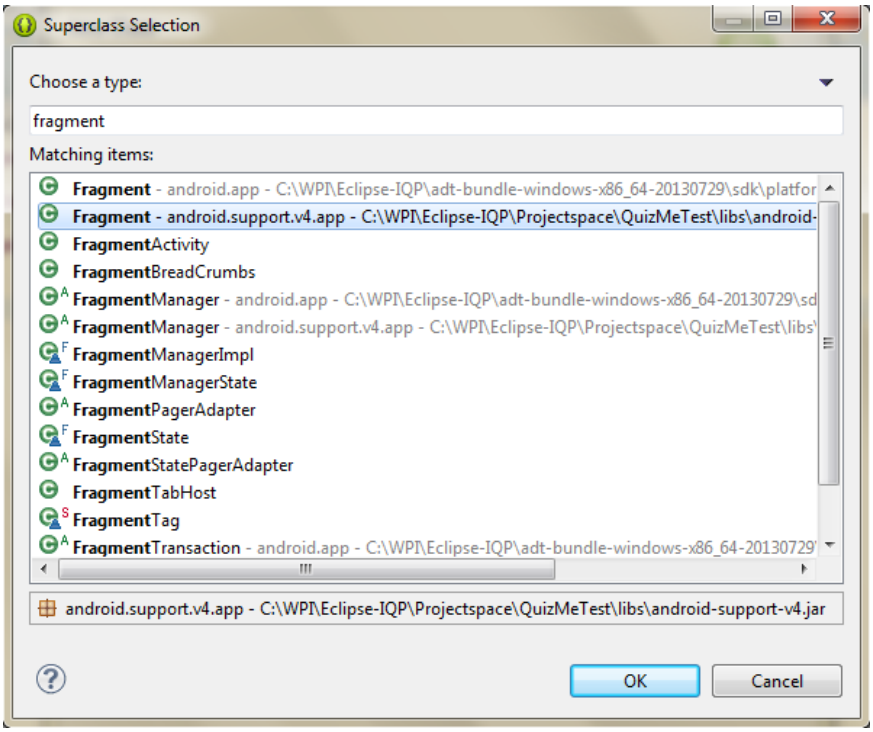

Figure 4.22: Superclass selection

Now that this class is created, go to the *res→layout* folder, right-click on *layout*, and select **new→Android XML File**. Name this file "fragment_run_truefalse" (Figure 4.23).

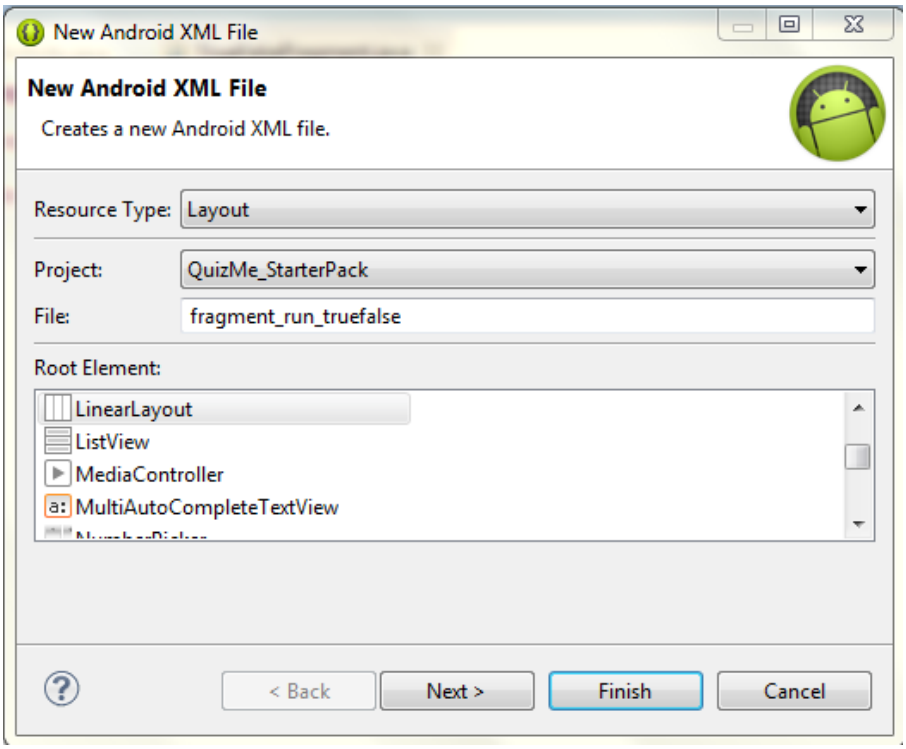

Figure 4.23: Manual layout XML file creation

This will be what appears in your fragment's area. In *fragment_run_truefalse.xml*, edit the code as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:Android="http://schemas.android.com/apk/res/android"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:gravity="center"
      android:orientation="vertical" >

      <TextView
            android:id="@+id/question_display"
            android:layout_width="wrap_content"
            android:layout_height="0dp"
            android:layout_weight="0.90"
            android:gravity="center|top" />

      <TextView
            android:id="@+id/answer_display"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="0.20"
            android:gravity="center"
            android:textAppearance="?android:attr/textAppearanceLarge" />
```

```
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:gravity="center">

        <Button
            android:id="@+id/answer_button1"
            android:layout_width="125dp"
            android:layout_height="75dp"
            android:text="@string/answer_button1" />

        <Button
            android:id="@+id/answer_button2"
            android:layout_width="125dp"
            android:layout_height="75dp"
            android:text="@string/answer_button2" />
    </LinearLayout>

</LinearLayout>
```

Add the new strings from the layout code above to *strings.xml*. Then go back to *TrueFalseFragment.java* as it is time to program the fragment's behavior. The first thing to note is that since fragments do not use Intents, a different method must be used to pass the data we wish to display in the fragment (the Question's text and the correct answer). To do this, a new function is defined inside the fragment to pass data is the same manner as an extra:

```
public class TrueFalseFragment extends Fragment {

    public static final String EXTRA_TF_QUEST_TEXTS =
"edu.wpi.it270x.quizme.tftext";
    public static final String EXTRA_TF_QUEST_ANS =
"edu.wpi.it270x.quizme.tfans";

    @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup parent,
Bundle savedInstanceState) {
      // Inflate the view
      View v = inflater.inflate(R.layout.fragment_run_truefalse, parent,
false);

      return v;
    }
```

```
    public static Fragment newInstance(String text, String ans) {
      Bundle args = new Bundle();
      args.putSerializable(EXTRA_TF_QUEST_TEXTS, text);
      args.putSerializable(EXTRA_TF_QUEST_ANS, ans);

      TrueFalseFragment fragment = new TrueFalseFragment();
      fragment.setArguments(args);

      return fragment;
    }
}
```

First, we define two more Extras as we have done previously. onCreateView inflates the fragment, although we will be adding to it so that it functions how we want it to. Next, we have newInstance. This newInstance method accepts a Question's text and answer, creates serializable values (essentially extras for fragments), and returns the fragment. Whenever a True or False Question fragment is desired, it will be obtained through this newInstance method. The **Bundle** class used in this method is essentially a collection of the extras to be passed through.

Now, the behavior of the fragment will be defined inside the method onCreateView. The code below is fairly large, but fairly simple. Explanations for each individual piece of this code have been written in comments within the code. Add the code in bold below to this class to create the graphical user interface for this fragment:

```
      public View onCreateView(LayoutInflater inflater, ViewGroup parent,
Bundle savedInstanceState) {
       // Inflate the view
      View v = inflater.inflate(R.layout.fragment_run_truefalse, parent,
false);

      // Set fields to hold values from the question
      String qtext = (String)
getArguments().getSerializable(EXTRA_TF_QUEST_TEXTS);
      final String qanswer = (String)
getArguments().getSerializable(EXTRA_TF_QUEST_ANS);
      Button true_button = (Button) v.findViewById(R.id.answer_button1);
      Button false_button = (Button) v.findViewById(R.id.answer_button2);

      final TextView question_display = (TextView)
v.findViewById(R.id.question_display);
      final TextView answer_display = (TextView)
v.findViewById(R.id.answer_display);
      question_display.setText(qtext);

      // Set the True and False answer buttons
      true_button.setOnClickListener(new OnClickListener() {
```

```
            @Override
            public void onClick(View arg0) {
                  // Check the answer given by the user compared to the
stored answer
                  // and print respectively
                  if (qanswer.equals("T")){
                        answer_display.setText("CORRECT!");
                  }
                  else {
                        answer_display.setText("INCORRECT!");
                  }
            }
      });
      false_button.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View arg0) {
                  // Check the answer given by the user compared to the
stored answer
                  // and print respectively
                  if (qanswer.equals("F")){
                        answer_display.setText("CORRECT!");
                  }
                  else {
                        answer_display.setText("INCORRECT!");
                  }
            }
      });
      // Return the view
      return v;
}
```

As you can see the fragment behaves like an Activity. The major difference is that fragments can be swapped out for other fragments during runtime. This will allow us to display multiple Question types later. Now that the fragment works, we must edit *RunQuizActivity.java* to incorporate fragments:

```
public class RunQuizActivity extends FragmentActivity {

    public static final String QUID = "edu.wpi.it270x.quizme.passedquiz";
    private int quiz_id;
    DatabaseHelperQuestions qdb;
    private int current_question;
    private int length;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_run_quiz);
```

```
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is
present.
        getMenuInflater().inflate(R.menu.run_quiz, menu);
        return true;
    }
}
```

This Activity is fairly simple, as we are setting up basic functionality and filling in most of the space with fragments. Setup is complete and onCreate will now be programmed to run the Quiz by hosting and swapping out fragments. Before *RunQuizActivity.java* is modified further, however, we must consider how to create the Question fragments. Because we will have multiple Question classes, we will again make the Question classes responsible for preparing their own fragments. In *IQuestions.java* add the method `public Fragment prepare_Fragment()`. Then implement this method in each Question class. For fill blank and multiple choice Questions, this method should return null for now. In *TrueFalseQuestion.java* have the method run thusly:

```
@Override
public Fragment prepare_Fragment() {
      return TrueFalseFragment.newInstance(this.questText, this.answer);
}
```

Now that True or False Questions can prepare their own fragments, go back to *RunQuizActivity.java* and prepare to add to the onCreate method. Again, this is a very large block of code, but its functionality is actually fairly basic. For easier understanding, we have again included comments in this code for reference so as to explain the various sections.

```
      protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_run_quiz);

       qdb = new DatabaseHelperQuestions(this);

       final FragmentManager fm = getSupportFragmentManager(); //This hosts
the fragments
       Fragment fragment = fm.findFragmentById(R.id.fragmentContainer);

       // Define buttons for navigation
       Button back_button = (Button) findViewById(R.id.back_button);
       Button next_button = (Button) findViewById(R.id.next_button);

       quiz_id = getIntent().getIntExtra(QUID, -1);
       // Error checking
```

```
        if (quiz_id == -1){
              finish();
        }

        //Get list of questions and convert to array
        LinkedList<I_Questions> available_questions =
qdb.getQuizQuestions(quiz_id);
        // Exit the Activity if there are no questions. This should not
happen, but just in case.
        if(available_questions.isEmpty()){
              finish();
        }
        else{
              // Set the array of current questions, beginning with 0
              current_question = 0;
              int i = 0;
              final I_Questions arrayofquestions[] = new
I_Questions[available_questions.size()];
              for(I_Questions tmp1: available_questions){
                    arrayofquestions[i] = tmp1;
                    i++;
              }

              i = 0;
              length = arrayofquestions.length;

              // Prepare the fragment of the first question
              if (fragment == null) {
                   fragment = arrayofquestions[i].prepare_Fragment();
                   fm.beginTransaction()
                   .add(R.id.fragmentContainer, fragment)
                   .commit(); //tell the fragment to appear
             }
              // Set the back and next buttons to navigate through the array,
              // coming back to the start if the end of the array is reached
              next_button.setOnClickListener(new OnClickListener() {
                    @Override
                    public void onClick(View arg0) {
                          current_question++;
                          if (current_question == length){
                                current_question = 0;
                          }
                          // Replace the fragment

fm.beginTransaction().replace(R.id.fragmentContainer,
arrayofquestions[current_question].prepare_Fragment()).commit();
                    }
              });
              back_button.setOnClickListener(new OnClickListener() {
```

```
                    @Override
                    public void onClick(View arg0) {
                          current_question--;
                          if (current_question == -1){
                                current_question = length - 1;
                          }
                          // Replace the fragment

fm.beginTransaction().replace(R.id.fragmentContainer,
arrayofquestions[current_question].prepare_Fragment()).commit();
                    }
              });
        }
    }
```

Testing the Application

Finally, we should have a functional app! Run QuizMe in the emulator as you have been doing and stress test it. Create several new Quizzes, try creating some with the same names, delete some quizzes, and create some Questions in several quizzes and try running them both. Answer both True and False to each Question to ensure those are working. In short, you want to try to break the application in any way you can. If you find something that does not seem to be working properly if a certain condition is met, go back and edit that Activity and fine tune your methods to check for that condition. If the application crashes, check the LogCat and find where the issue lies.

Congratulations! You have developed your first Android application. We are not done yet, however; QuizMe in its current state leaves a lot to be desired. We should allow for more than just True or False Questions, and providing the user with more feedback would be helpful. When all is said and done, you will want to clean up your code.

4.10: Multiple Choice and Fill-in-the-Blank Questions

Overview

As mentioned, even though QuizMe is technically usable, it could use a lot of functionality improvements. We already have fragments implemented, so we should implement multiple Question types first. Our final application will support True or False, Multiple Choice, and Fill-in-the-Blank Questions. The database will have to know which type of Question is being created, updated, or run in a Quiz, and the proper Activity or fragment will have to be loaded. Of course, in addition to the fragments, we will require an additional Activity for adding or editing each type of Questions. Several activities we already have will have to updated to accommodate these new types of Questions.

Multiple Choice Questions will consist of a Question and four answers, each with a radio button to designate the answer. Save and Discard buttons should be present during editing, and there should be a Submit Answer button when answering one of these Questions (Figure 4.24).

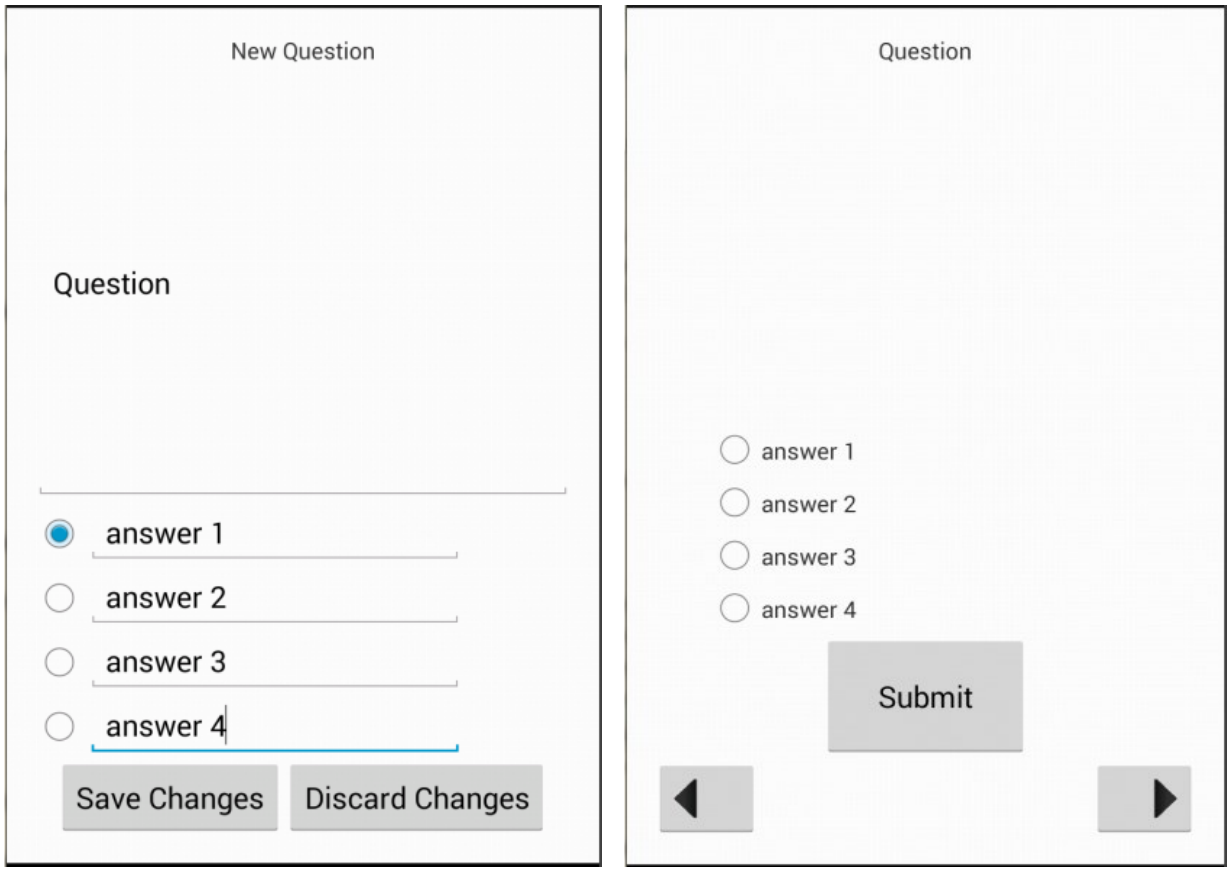

Figure 4.24: Activities handling multiple choice Questions

Fill-in-the-Blank Questions will consist of two text fields, one for a Question and one for an answer (the type of text field, TextView or EditText, depends on whether a Question is being run or edited of course). Save and Discard buttons should be present during editing, and there should be a Submit Answer button when answering one of these Questions (Figure 4.25).

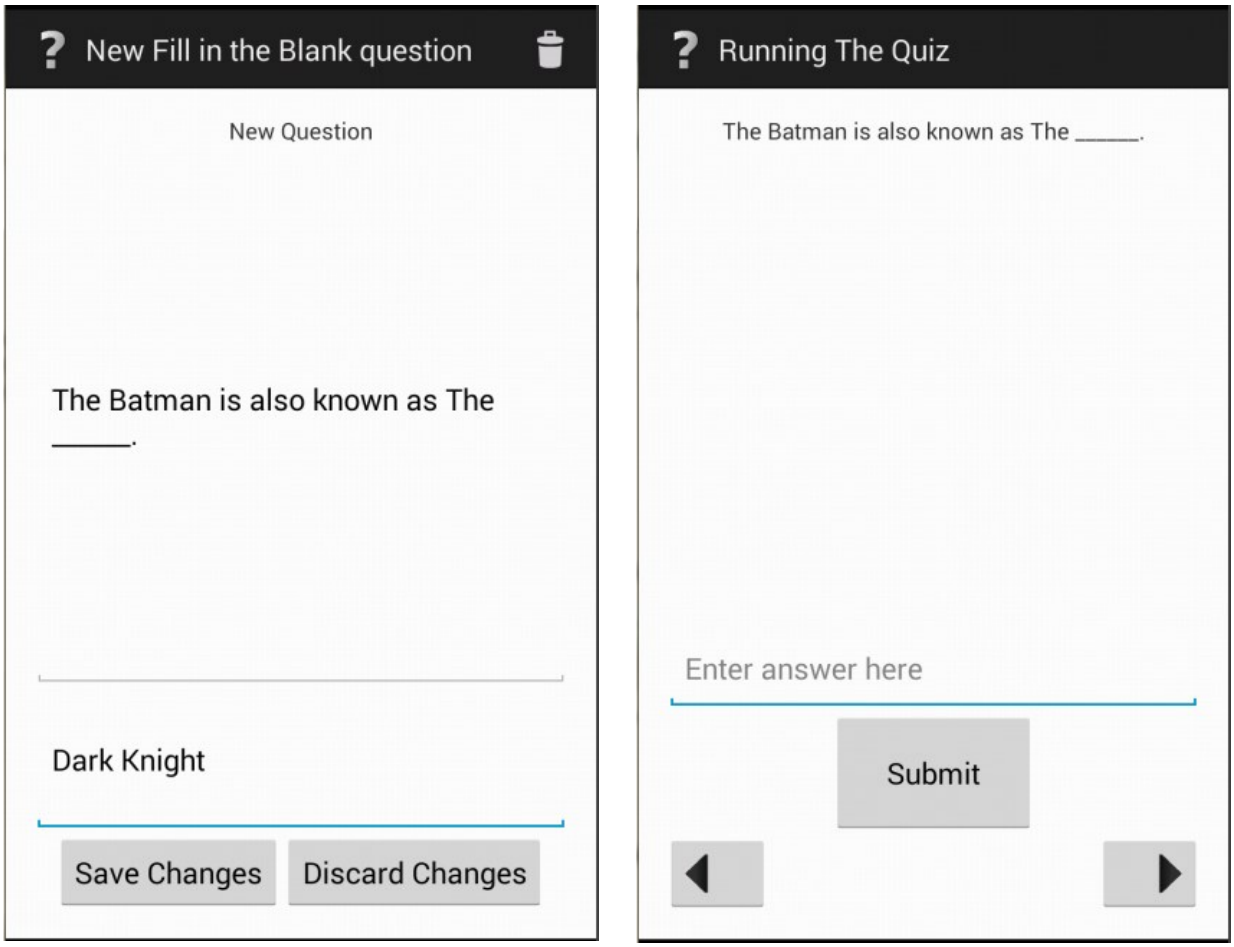

Figure 4.25: Activities handling Fill-in-the-Blank Questions

We highly recommend starting with Fill-in-the-Blank Questions, but this choice is entirely yours.

Development: Fill-in-the-Blank

To implement Fill-in-the-Blank Questions you will need to:

- Create an Activity to edit Fill-in-the-Blank Questions
- Fill in *FillBlankQuestion.java*'s prepare_Editor method
- Create a fragment to run Fill-in-the-Blank Questions
- Fill in *FillBlankQuestion.java*'s prepare_Fragment method
- Add the option to create Fill-in-the-Blank Questions to the menu of *QuizListActivity*

Development: Multiple Choice

To implement multiple choice Questions you will need to (it is highly recommended you research Android radio buttons, as there can be some tricky logic here; this will be a challenge, be sure to test thoroughly!):

- Create an Activity to edit multiple choice Questions

- Fill in *MultChoiceQuestion.java*'s prepare_Editor method
- Create a fragment to run multiple choice Questions
- Fill in *MultChoiceQuestion.java*'s prepare_Fragment method
- Add the option to create multiple choice Questions to the menu of *QuizListActivity*
- Note: when one radio button is selected, all others should be unselected

Note that for multiple choice Questions, the official version of the QuizMe application supports a minimum of two answers and a maximum of four. If there are less than four, the unused radio buttons and fields will not appear when running the Quiz.

4.11: Minor Improvements and Cleanup

Congratulations, QuizMe is now fully functional, and much more useful than the last time we said that! The last thing to do will be to add some additional features and to clean up your code.

Color Feedback

In software development, the more feedback you can provide the user without overloading them, the better. As such, you should edit the “CORRECT!” and “INCORRECT!” text to be green and red, respectively.* The Android Developers documentation should tell you everything you need to know, and it is your task to find out how to do this.

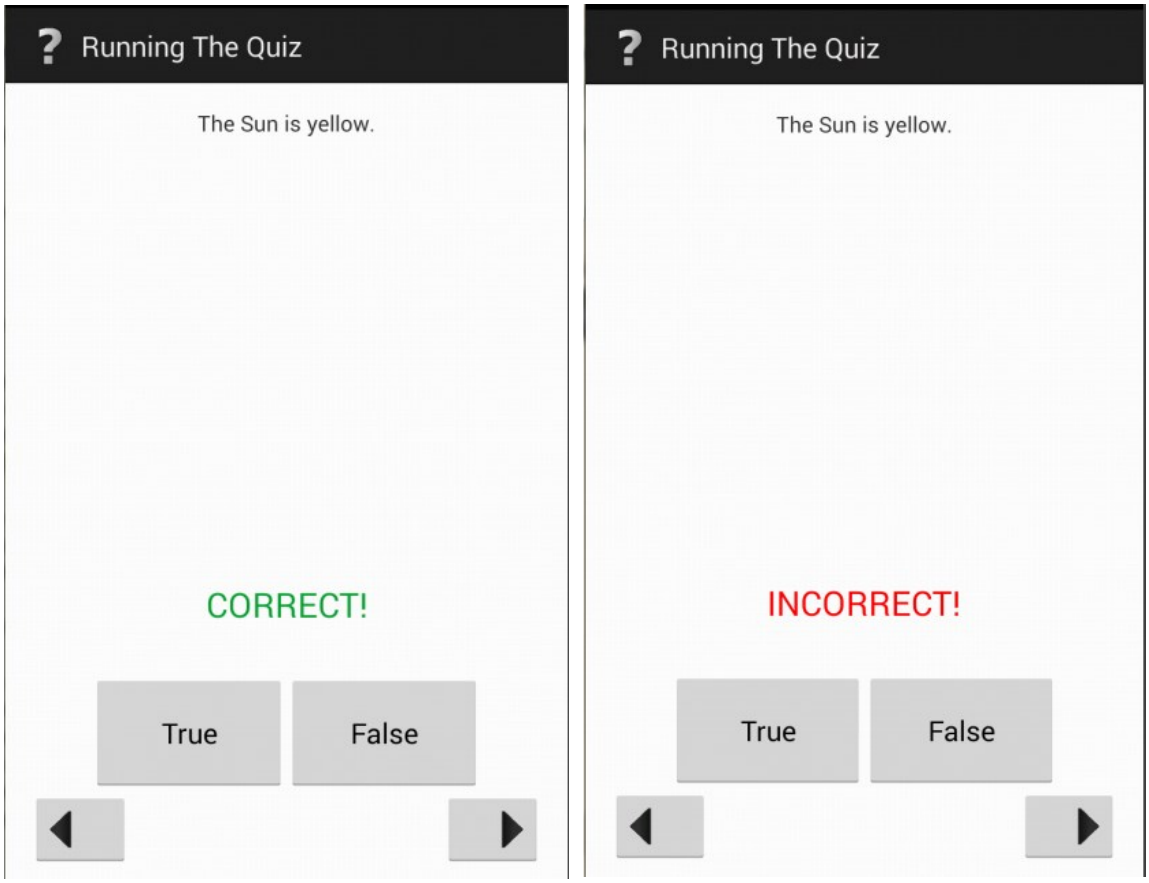

Figure 4.26: Colored answer feedback

You may notice that the default `Color.green` seems to be too bright. “Red” is fine, but for the green color, you should use the hex code for a darker green (we used #0AA336). You will need to find out how to parse a hex color code, as simply entering `Color.#0AA336` or something similar will not work (Figure 4.26).

Toast Popups

If you’ve ever used an Android device regularly, you most likely have seen toast popups before. These are very useful for reporting feedback to the user. Your task will be to create the toast shown in the figure below. This toast should pop up if a user tries to run a Quiz that has no Questions associated with this Quiz’s ID (Figure 4.27). As well, you should place toasts into all three Question editor activities so that if a Question does not meet the minimum requirements, the user is informed of the error if they try to save this Question. If you can think of any other part of the application that may be improved with a toast, feel free to add it in; this is highly encouraged.

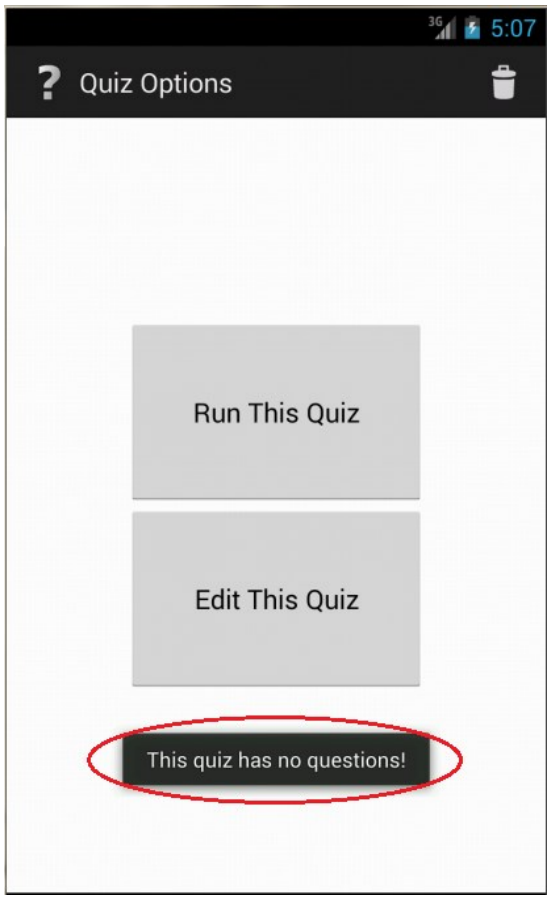

Figure 4.27: A Toast popup on QuizOptionsActivity

Code Cleanup

Now that all of the required features and functionality are in place, you should go through your code and clean it up. First, find any warnings about unused code and imports and remove this “dead” code. With that code left in place, the application is still creating those methods and calling those imports, yet they are never used, so performance can suffer slightly.

You may notice several warnings in the XML files. Unless these are referring to “dead” code, these can be ignored if everything is displaying properly.

Throughout your code, you may also encounter deprecation warnings. Including deprecated functions and techniques is not typically good practice, but the reasoning varies by function. You may leave these in place, but please be aware in the future that if this warning appears, you should find a more up-to-date alternative to whatever function you are calling, as a deprecated function typically has either security vulnerabilities or performance flaws.

If you wish, you could go through and refactor any functions and classes that you feel should have more descriptive names. To refactor a function or class, right-click on its name and click **Refactor** → **Rename**. Refactoring will change this name throughout the Java code, **but not in the XML code.** If you do decide to refactor anything, ensure that no references to the old names are present in the XML.

You should now go through your Java code and improve your commenting. Even if you kept up with your comments during development, it is still good practice to update them before you turn in the final product. Add comments where you are lacking any, remove or update outdated and unneeded comments, and ensure that any tricky parts of the code are fully explained in comments.

Finally, run the application in the emulator once again and test the app. In particular, in the development team's creation of these applications, we encountered many unexpected issues with multiple choice Questions, as well as whitespace and mixed letter case in Fill-in-the-Blank Questions, so be sure to test these thoroughly. If you encounter any issues, track down the cause and do what you can to fix these bugs.

Once you are sure your application is bug free, export the application as a .zip file. If you wish, you may also create an APK to install on your own Android device (that is, if you haven't been testing on an actual device already, in which case, testing it on your device one last time will install the most recent version). An APK file is a semi-compiled compressed file. It contains a compiled android application along with metadata needed to install said application on a device. To get the actual APK file of your application you can run it on the emulator. After it has successfully run on the emulator you can find an APK file ***Application****.apk* in the "bin" directory of your project.

To export the project right-click it and select "Export" then choose the "Archive File" option. Select the project and destination and click finish (Figure 4.28).

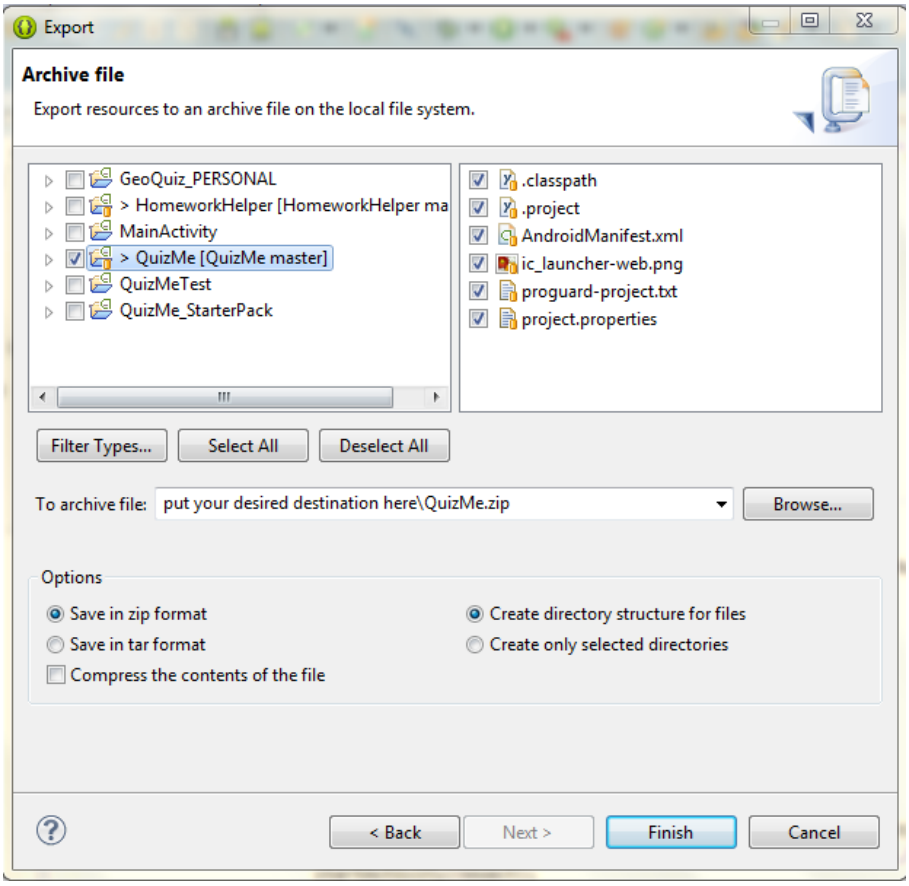

Figure 4.28: Exporting QuizMe as a .zip archive file

Congratulations, you have finished your first Android application development project!
Now, we will move on to a more advanced project, HomeworkHelper.

Chapter 5 - HomeworkHelper (Instructional)

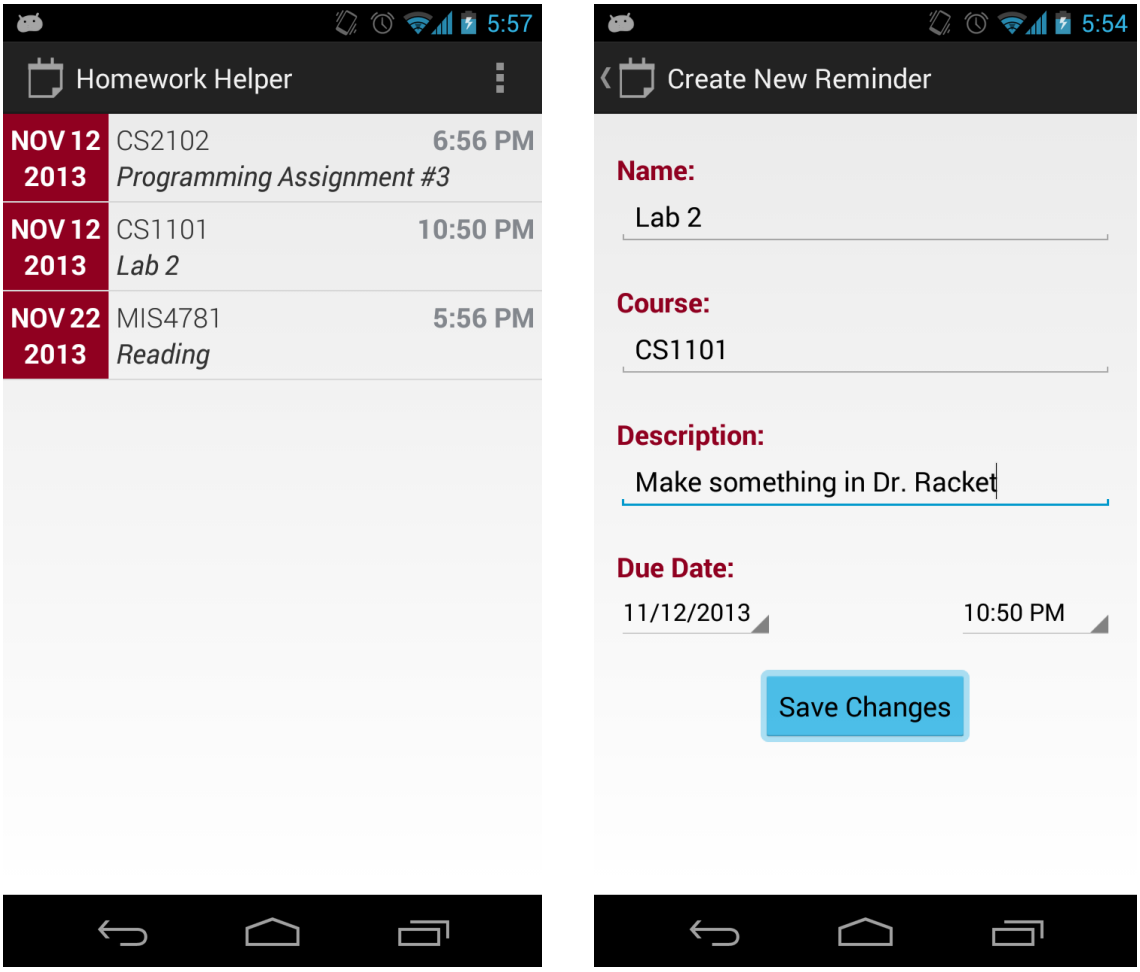

Figure 5.1: The HomeworkHelper app

5.1: Environment Setup

To set up the project, extract and import the *HomeworkHelper_StarterPack.zip* file provided by your instructor into the Eclipse workspace following the same method used by the Stub App and QuizMe Apps. Please note that you will be using a different file instead of the Stub App because of how much different this application is to QuizMe. It would take us much more time to walk you through making unnecessary changes to the Stub App in order to make it work with this, so we just decided to give you a starter package to build off of.

5.2: Building the Application: Overview

This application will act as a daily planner, allowing students to set reminders for themselves for their coursework at WPI. Similar to the QuizMe application in the previous section, this application will store all the necessary data in a SQLite database stored on the Android device. Unlike the QuizMe app, we will be adding many more features to this application such as support for notifications, automated SQL interaction functions, and transfer of information to other applications. We will use this transfer of information to automatically fill in

certain fields on the built-in Calendar application provided by the Android operating system. We will also add a preferences section to allow students to enable or disable the features mentioned.

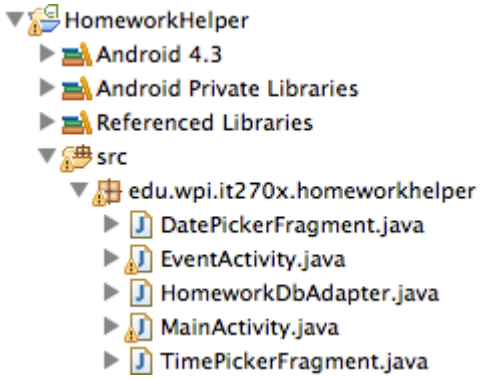

Figure 5.2: The Eclipse project manager

Unlike the QuizMe project, you will not be modifying the Stub App to create HomeworkHelper. Instead, you will import the project directly. Reference the Stub App instructions if you are having trouble with the import. This method is more efficient for this project, and you already have experience using the refactoring method with QuizMe. After you import the project into your Eclipse workspace, you will notice a few things similar to Figure 5.2. First of all, there are only two Activities. We will be creating another one soon for a total of three. There are also two Fragments. Finally, there is a class called *HomeworkDbAdapter*. This will be where we store our SQLite database and all the appropriate functions we will use in this application. These are provided for you. Let's begin modifying HomeworkHelper by making a new Activity called *Splash Screen*.

5.3: Splash Screen

Overview

A Splash Screen is not necessary to the application, but it provides an aesthetically pleasing feature. HomeworkHelper's Splash Screen will display the WPI logo for a number of seconds, then proceed to the Main Activity. Later in this chapter, we will configure this application to enable or disable this functionality by creating an Activity for preferences. Unlike QuizMe's Splash Screen, HomeworkHelper's will fade away automatically and will not accept any user input.

Development

In order to create this Splash Screen, we will first need to create a new Android Activity. To do this, in Eclipse go to **File** → **New** → **Other** and select **Android Activity** from the Android folder as shown in Figure 5.3 below:

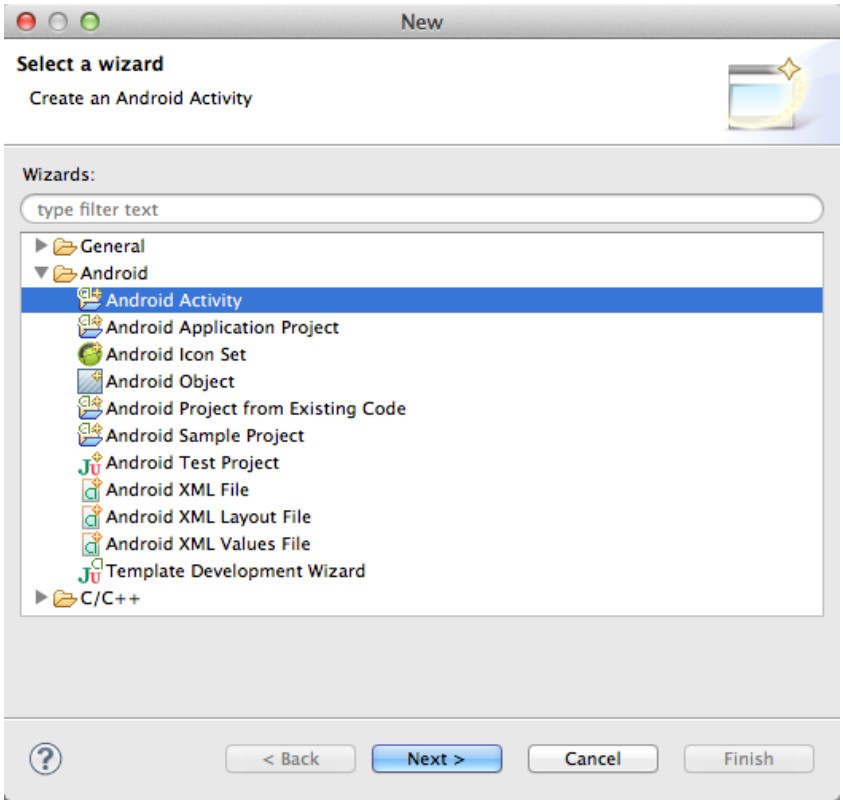

Figure 5.3: Create Activity

Click the **Next** button and select **Blank Activity** as shown in Figure 5.4.

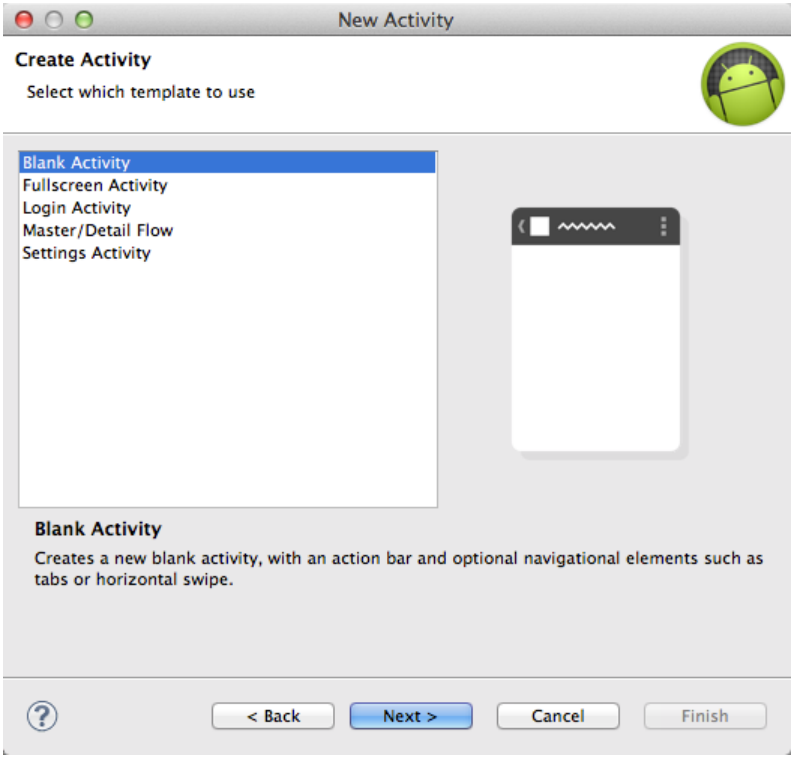

Figure 5.4: Creating a blank Activity

Click the **Next** button and be sure that the following apply in the next screen:

Project: make sure **HomeworkHelper** is selected
Activity Name: “Splashscreen”
Layout Name: “activity_splashcreen”
Title: “Splashscreen”

After you have typed these values in, your window should look like Figure 5.5.

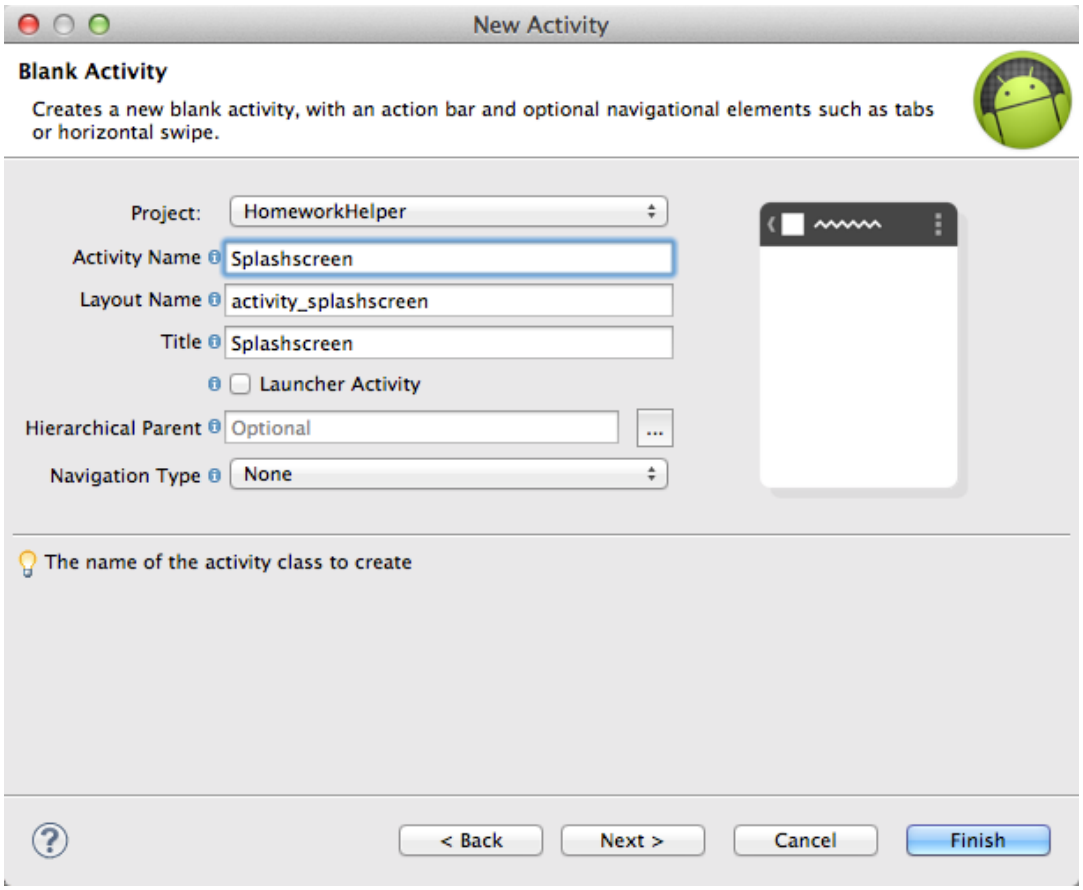

Figure 5.5: Name the Activity

Select **Finish** to create the new file. In the *Splashscreen.java* file, replace the existing class code, underneath the package declaration and the imports, with the following:

```
public class Splashscreen extends Activity {
    // Splash Screen timer
    private static int SPLASH_TIME_OUT = 3000;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_splashscreen);
    }

    @Override
    protected void onResume()
    {
        super.onResume();
        new Handler().postDelayed(new Runnable()
            {
                @Override
                public void run()
                {
                    //Finish the splash activity so it cannot be returned to.
                    finish();
                    // Create an Intent that will start the Main Activity.
                    Intent mainIntent = new Intent(Splashscreen.this,
MainActivity.class);
                    startActivity(mainIntent);
                }
            }, SPLASH_TIME_OUT);
    }
}
```

The static int created, **SPLASH_TIME_OUT**, provides the length of time for the Splash Screen to run. Here we specify how long the splash screen will display – in this case for 3000 milliseconds which is equal to 3 seconds. Then, the **onCreate(Bundle savedInstanceState)** method is called, which inflates the Splash Screen Activity. Finally, the **onResume()** method is called, which nests the **run()** method, which uses an Intent to start the Main Activity, and the Splash Screen ends.

Save these changes and import the necessary packages as you have done previously. If you are still seeing some red underlined words, double check to make sure you have the imports shown here at the top of this file:

```
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.os.Handler;
```

Then, be sure to modify the *AndroidManifest.xml* file (as shown in Figure 5.6) to include the following code above the Main Activity section:

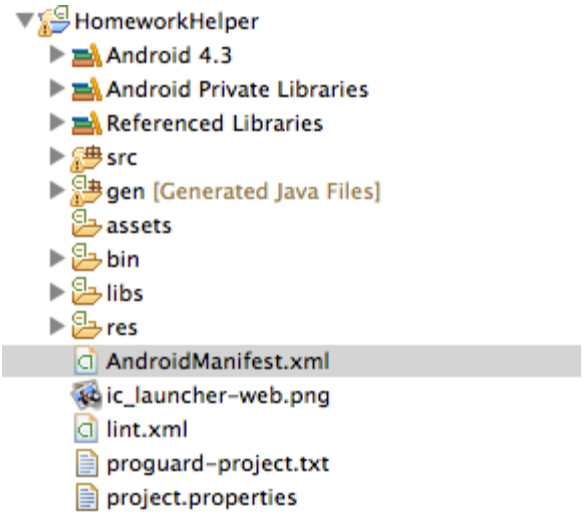

Figure 5.6: Locating AndroidManifest.xml

This file is one of the more important files in Android development because it ties the entire scope of the application. We will come back to this file later.

```
...
<application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >

        <!-- Splash Screen -->
        <activity
            android:name=".Splashscreen"
            android:configChanges="orientation|keyboardHidden|screenSize"
            android:screenOrientation="portrait"
            android:label="@string/app_name"
            android:theme="@android:style/Theme.Holo.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <!-- Main Activity -->
```

Save your changes. Eclipse may complain about an error here. This is because when an Activity is created, the Android Manifest file is automatically updated. We manually edited it so that we could include more features, such as making the application launch with the Splash Screen rather than the Main Activity. To clear these errors, delete this block of code near the bottom of *AndroidManifest.xml*:

```
        <activity
            android:name="edu.wpi.it270x.homeworkhelper.Splashscreen"
            android:label="@string/title_activity_splashscreen" >
```

```
</activity>
```

Next, we need to edit the Splash Screen layout. Navigate to the *res* → *layout* folder and modify the *activity_splashscreen.xml.* Replace everything in the file with the following code:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#000000" >

    <ImageView
        android:id="@+id/imgLogo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:contentDescription="@string/wpi_logo"
        android:src="@drawable/wpi_logo" />
</RelativeLayout>
```

Let's review what we've done so far. At this point, you have successfully:

- Made a new Activity
- Set the layout of the Activity by updating the *activity_splashscreen.xml* file
- Manually updated the *AndroidManifest.xml* file to reflect that the application now has a new Activity

Try running the application. You'll notice that the Splash Screen Activity that you just made comes up for about 3 seconds then goes right into the main screen of the program. This is exactly what is expected, so if you have gotten this far you are doing well.

5.4: Event Activity

Overview

If you ran the application after making the changes in the previous section, you will notice that our application does not really do much at this point and that there is no real connection between all the Activities you have available. So far our application only loads up a Splash Screen we made and then goes into the Main Activity, in which we cannot currently do anything. You might be asking, how do we get things to display in the Main Activity? The answer is in the Event Activity, which we will work on now. Refer back to the screenshot in Figure 5.1 to see what this Activity will look like.

Development

We will start by creating the layout. To begin, open up the *activity_event.xml* file in *res* → *layout* folder. You should see the following code:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="15dp"
    android:paddingRight="15dp"
    android:orientation="vertical" >
```

```
</LinearLayout>
```

This **LinearLayout**, is used in Android is just as the name suggests. Elements of the user interface are laid out linearly, in the order that you put them in. The code above sets the height and width to *match_parent*, in this case filling the whole screen. We then set the orientation to *vertical* and apply padding on the left and right side so that the content does not go right up to the edge of your Android device.

Let's start building what the screen should look like. Keep in mind that the order of these code blocks is what determines the order in which they are displayed in the application. Add the following code within the **LinearLayout** block:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="25dp"
    android:text="@string/event_name"
    android:textSize="18sp"
    android:textColor="#900020"
    android:textStyle="bold" />
```

The code above adds a margin from the top using the `android:layout_marginTop` attribute. We also give the **TextView** a custom font size using `android:textSize`, font style using `android:textStyle`, and font color using `android:textColor` for a more aesthetically pleasing interface.

The **EditText** we are going to add is below the "event_name" **TextView** and enables the functionality of editing the event name:

```
<EditText
    android:id="@+id/edit_event_name"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/edit_event_name_placeholder" />
```

Now add a **TextView** that will display the string "Course:" by referencing the `event_course` string in the *strings.xml* file below the previous **EditText**.

The **AutoCompleteTextView** is another predefined Android feature that provides text suggestions to the user while they are editing the course name.

```
<AutoCompleteTextView
    android:id="@+id/edit_event_course"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/edit_course_placeholder" />
```

The code above acts very similar to what an **EditText** does in Android. The only exception is that, after you input a few characters, it will automatically give you a list of options that you can select from. If you have ever filled out forms online, you may have already seen this when visiting a web page that you have already filled out information for. In context, the **AutoCompleteTextView** will be filled in with a list of courses at WPI, which we will add later.

Now, add a **TextView** that will display the text “Description:” and the appropriate **EditText**. Add another **TextView** that will display the text “Due Date:”

Now add the following code below. We will explain what this does shortly.

```
<RelativeLayout
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:orientation="horizontal" >

      <TextView
            android:id="@+id/pick_date"
            style="?android:attr/spinnerStyle"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentLeft="true"
            android:layout_alignParentTop="true"
            android:hint="@string/event_select_date" />

      <TextView
            android:id="@+id/pick_time"
            style="?android:attr/spinnerStyle"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentRight="true"
            android:layout_alignParentTop="true"
            android:hint="@string/event_select_time" />
</RelativeLayout>
```

The code above wraps two **TextViews** in a **RelativeLayout**. This is different from a **LinearLayout** in the sense that elements are laid out relative to one another instead of linearly. This allows us to put two elements - the **TextViews** - side by side. If we used a **LinearLayout** one of the textboxes would be on top and the other on the bottom. **Both TextViews** are styled using the `style` attribute. They are made to look like **Spinners**, another Android object that is used for displaying a list of things to select. **Spinners** are very similar to combo boxes that allow you to select an option, rather than having to type something in. Yet, we will be using these **TextViews** for something else that we will cover soon.

Finally, add a “Save Changes” button using the following code:

```
        <Button
            android:id="@+id/btn_save"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:layout_marginTop="20dp"
            android:text="@string/save_changes" />

</LinearLayout>
```

Save the changes made to the *activity_event.xml* file and verify the correct changes were made using the Graphical Layout tab. Your Activity should look similar to Figure 5.7:

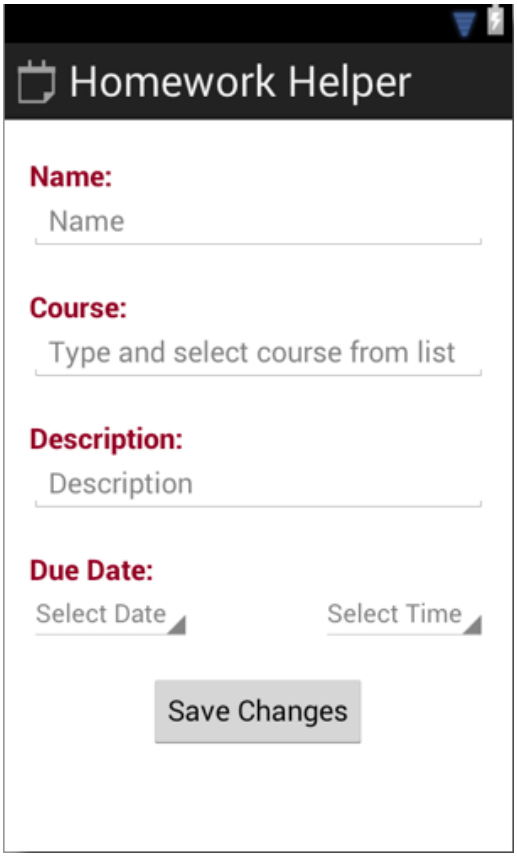

Figure 5.7: Event Activity

At this point, you have completed most of the layout of the Event Activity by specifying what it should *look* like. Let's try running the application again. Notice anything different? Event Activity is never used! We can already get from the Splash Screen to the Main Activity, but how do we get from the Main Activity to Event Activity? We first have to modify *MainActivity.java* in order to do this. Let's start by opening up this file and adding this function between the `fillData()` and `createReminder()` methods:

```
    private void fillData(){
    …
    }
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is
present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
    public void createReminder(){
```

. . .

Chances are that `R.menu.main` will throw an error. We will fix that right now by right clicking on the *res* → *menu* folder as shown in Figure 5.8 and selecting **New** → **Android XML File**.

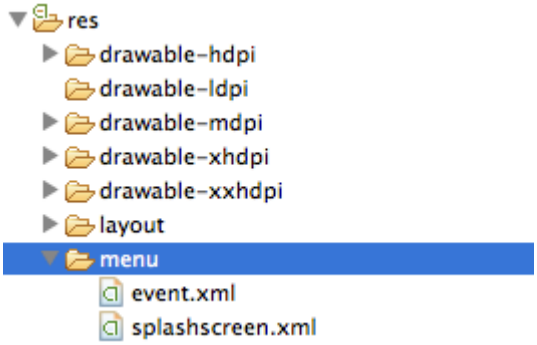

Figure 5.8: Menu Location

You should then see something similar to Figure 5.9 below.

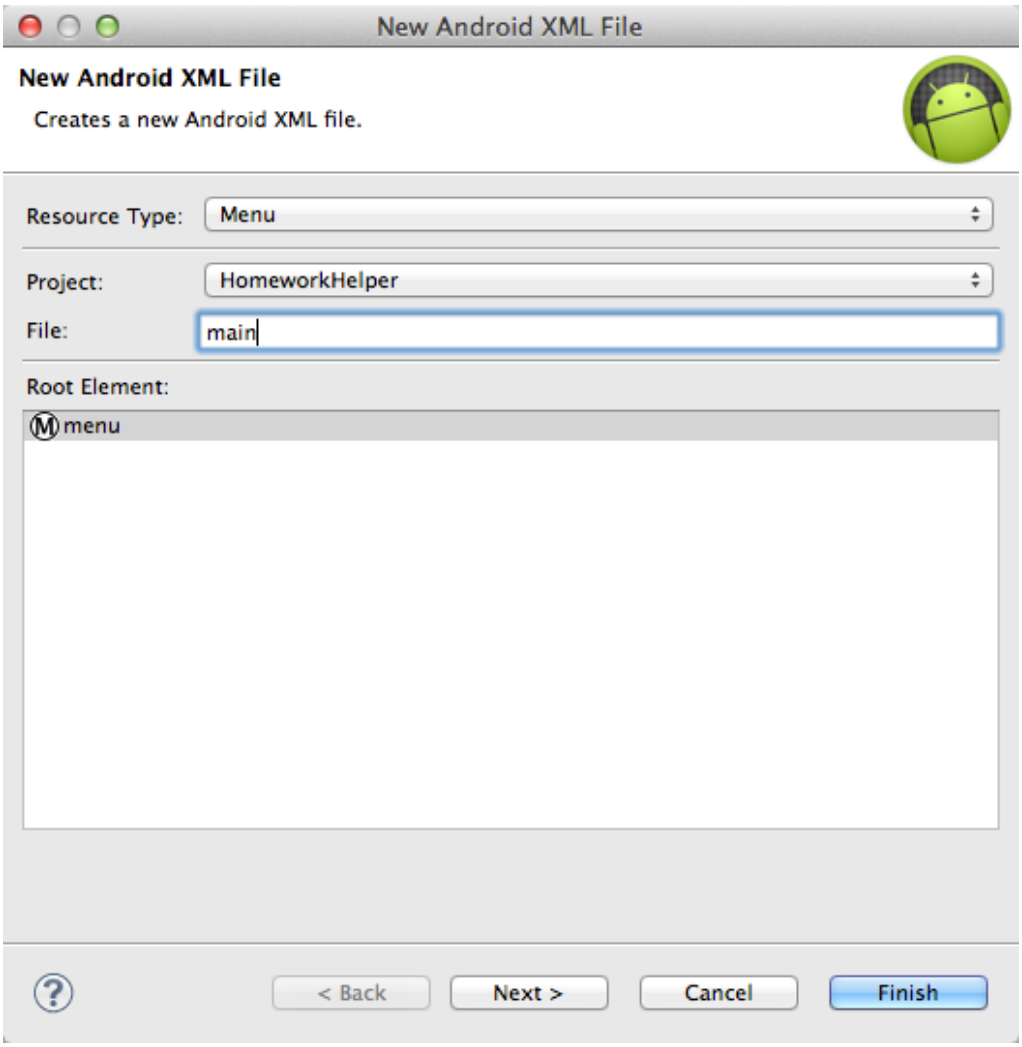

Figure 5.9: Creating the main.xml file

Type in the word “main” next to File and click the **Finish** button. The *main.xml* file you just created should load up and look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >


</menu>
```

Now add the following code in **bold**:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/action_create"
        android:showAsAction="never"
        android:title="@string/action_create" />

    <item
```

```
        android:id="@+id/action_settings"
        android:showAsAction="never"
        android:title="@string/action_settings"/>

</menu>
```

By doing this, we just added two options for when a user presses the Menu button. Save the changes and try running the application. When running this code, your menu should look like Figure 5.10 below:

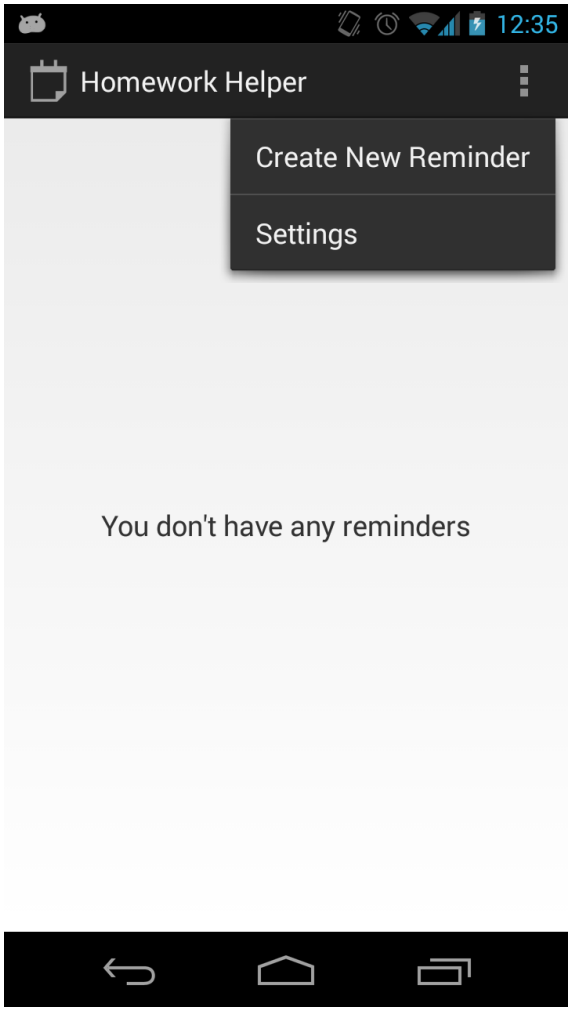

Figure 5.10: Menu Items

Try tapping on both of these items. You will notice that nothing happens at this point. Let's change that by opening up the *MainActivity.java* file and adding this function below the `onActivityResult()` method:

```
@Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle item selection
```

```
        switch (item.getItemId()) {
            case R.id.action_create:
                createReminder();
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }
```

The code we just added checks to see if a menu item was *selected*. If so, find out which one it was and execute whatever code we want for each item. In this case, our menu item called "Create Reminder" has the id `action_create` which we then use to call the `createReminder()` function that we already made for you. Let's take a look at what that does:

```
    public void createReminder() {
     Intent i = new Intent(this, EventActivity.class);
     i.putExtra(REQUEST_CODE, ACTIVITY_CREATE);
     startActivityForResult(i, ACTIVITY_CREATE);
    }
```

At the start of this function, we use an Intent to start the *EventActivity.java* file. We add something to this Intent called *REQUEST_CODE* (something we will cover in a little bit) and then call the function `startActivityForResult()` using both the *REQUEST_CODE* and the *ACTIVITY_CREATE* integer which we declared on the top of the *MainActivity.java* file to be equal to 0.

Try running the application again. This time, if you press the Menu button and select "Create Reminder", the Event Activity we made gets loaded! What do we do from here?

Let's move back to the *EventActivity.java* file. We started off with the code below. Let's get into a deeper explanation of what each part does.

```
package edu.wpi.it270x.homeworkhelper;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;

public class EventActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_event);
```

```
	}

	@Override
	public boolean onCreateOptionsMenu(Menu menu) {
	// Inflate the menu; this adds items to the action bar if it is
present.
		getMenuInflater().inflate(R.menu.event, menu);
		return true;
	}
}
```

The `onCreate(Bundle savedInstanceState)` function is part of an Activity's lifecycle, as mentioned in the QuizMe application. It is meant to be called only once when an Activity is created and executed and its purpose is to save the Activity's previous state, so no information is lost when the Activity runs again. We can extrapolate this same definition for the `onCreateOptionsMenu(Menu menu)` function, for use of the options menu.

Right above the `onCreate()` function, let's start declaring variables that we will use throughout this file.

```
public class EventActivity extends FragmentActivity {

	private EditText mNameText; // 'Name' textbox
	private AutoCompleteTextView mCoursesText; // 'Courses' textbox
	private EditText mDescriptionText; // 'Description' textbox
	private TextView mDateText; // 'Date' box
	private TextView mTimeText; // 'Time' box
	private Button mSaveChanges; // 'Save Changes' button

	private Long mRowId; // number used for the row ID when storing in
SQL
	private HomeworkDbAdapter mDbHelper; // Database object we will be
using to add things to our database

	private int requestCode; // request code integer for
'creating'/'editing' a reminder
```

Be sure to import any necessary packages and then save your changes. Now add the following code to the `onCreate()` function:

```
protected void onCreate(Bundle savedInstanceState) {
	super.onCreate(savedInstanceState);
	setContentView(R.layout.activity_event);
	mDbHelper = new HomeworkDbAdapter(this);
	mNameText = (EditText) findViewById(R.id.edit_event_name);
	mCoursesText = (AutoCompleteTextView)
findViewById(R.id.edit_event_course);
```

```
        mDescriptionText = (EditText)
findViewById(R.id.edit_event_description);
        mDateText = (TextView) findViewById(R.id.pick_date);
        mTimeText = (TextView) findViewById(R.id.pick_time);
        mSaveChanges = (Button) findViewById(R.id.btn_save);

        // Create an ArrayAdapter using the string array and a default spinner
layout

        ArrayAdapter<CharSequence> adapter =
ArrayAdapter.createFromResource(this, R.array.courses_array,
android.R.layout.simple_spinner_item);

        // Specify the layout to use when the list of choices appears
        adapter.setDropDownViewResource(android.R.layout.simple_spinner_item);

        // Apply the adapter to the AutoCompleteTextView
        mCoursesText.setAdapter(adapter);
```

Import any necessary packages, save your changes, and let us explain what we just added. The first thing to look at is the `mDbHelper = new HomeworkDbAdapter(this)` line. This initializes the mDBHelper object we declared earlier at the top of the file. We will be using this object for everything with connecting our application to the stored database.

The next few lines initialize the all the user interface elements we made in the *activity_event.xml* file and matches them up by their ID that we gave them. The Android SDK already includes a predefined function called `findViewById()` which is exactly what we use here to match up our layout in the *activity_event.xml* file to our code in the *EventActivity.java* file.

The subsequent lines of code create an ArrayAdapter, which we will use to store all the strings in our courses array. If you don't remember seeing this, take a look at the *res → values → strings.xml* file and look for the following block:

```
        <string-array name="courses_array">
        ...
        <item>ECE2010</item>
        <item>ECE2019</item>
        <item>ECE2029</item>
        <item>ECE2049</item>
        <item>ECE2201</item>
        <item>ECE2112</item>
        <item>GE2341</item>
        ...
        </string-array>
```

Our ArrayAdapter goes through each of these elements, which allows you to type in any matching characters so that you can select the appropriate courses that are populated in the

AutoCompleteTextView. Try running the application and tap on the “Create Reminder” menu item. Once the Event Activity loads up, tap on the “Courses” text box and type in something like ‘CH’ or ‘MIS’. You’ll notice that after you type in any matching two characters, you will get a list of all courses that match your input at which point you can just select without having to type in the name of the full course. Please note that if the course name does not exist in our list of courses, no other options come up. Let’s go back to the *EventActivity.java* file and add the following code where we left off:

```
// Listener for 'Set Date' TextView
mDateText.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
        DialogFragment newFragment = new DatePickerFragment(mDateText,
requestCode);
        newFragment.setCancelable(false);
        newFragment.show(getSupportFragmentManager(), "datePicker");
        }
});

// Listener for 'Set Time' TextView
mTimeText.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
      DialogFragment newFragment = new TimePickerFragment(mTimeText,
requestCode);
      newFragment.setCancelable(false);
      newFragment.show(getSupportFragmentManager(), "timePicker");
      }
});
```

Be sure to import any necessary packages. If you are automatically importing them, be sure to select **android.support.v4.app.DialogFragment** when prompted to import the package for **DialogFragment**. After you save your changes, you may notice that you still have errors because of the `getSupportFragmentManger()` function. To fix this, change the first line of the *EventActivity* class declaration from this:

```
public class EventActivity extends Activity {
```

to this:

```
public class EventActivity extends FragmentActivity {
```

Import any missing packages and then save your changes. You should not have any more errors. The code above needed to be changed because of the **android.support.v4.app.DialogFragment** that we imported earlier. This specific package

allows us to use **DialogFragments** in Android operating systems older than 3.0. Therefore, we also had to change how the Event Activity functions to accommodate for this support.

Now take a look at the two functions we just added. Both of these functions create event listeners for the Date and Time **TextViews** we put up in our Event Activity. These "event listeners" wait for a user to click/tap on that item and then executes the appropriate code for when that happens. In context, when `mDateText` is tapped on it loads up a new **DialogFragment** that shows a **DatePicker**. Similarly, when `mTimeText` is tapped on it shows a **TimePicker**. To make things easier for you, we have already made these pickers for you in both *DatePickerFragment.java* and *TimePickerFragment.java* respectively.

Run the application and try tapping on both the 'Select Date' and 'Select Time' **TextViews** and watch what happens. It looks like our event listeners are working! When you tap on either, you should see something similar to Figure 5.11 below:

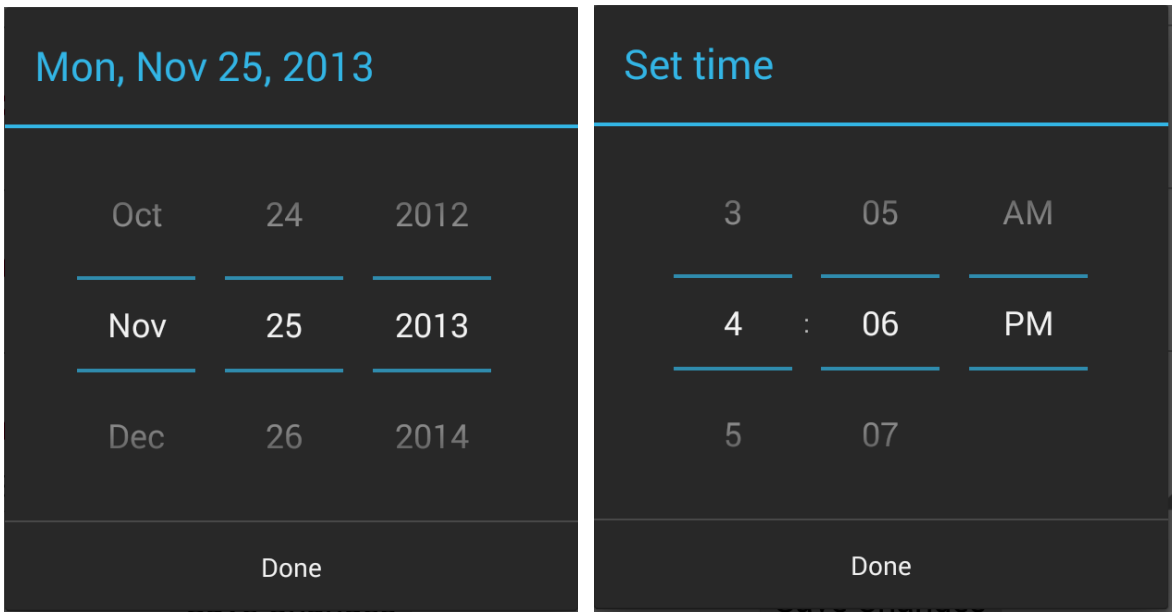

Figure 5.11: DatePicker and TimePicker Fragments

Try pressing the back button and you will notice that the fragments don't go away. We configured both of these so that a user *has* to press the 'Done' button due to a bug our team realized when the back button was pressed.

At this point, almost everything in the Event Activity has a purpose… except for the Save Changes button. Let's change that by opening up the *EventActivity.java* file and adding an event listener for `mSaveChanges`:

```
// Listener for 'Save Changes' button
mSaveChanges.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
            saveState();
            setResult(RESULT_OK);
            finish();
       }
});
```

```
} // end of onCreate() function
```

Save your changes. You will notice that the `saveState()` function is underlined. This is because we haven't made this method yet so let's go ahead and do that below the `onCreate()` function:

```
private void saveState() {
        String title = mNameText.getText().toString();
        String course = mCoursesText.getText().toString();
        String description = mDescriptionText.getText().toString();
        String date = mDateText.getText().toString();
        String time = mTimeText.getText().toString();

        if (mRowId == null) {
            long id = mDbHelper.createReminder(title, course, description,
date, time);
            if (id > 0) {
                mRowId = id;
            }
        } else {
            mDbHelper.updateReminder(mRowId, title, course, description,
date, time);
        }

        String[] aDate = date.split("/");
            int month = Integer.parseInt(aDate[0]) - 1;
            int day = Integer.parseInt(aDate[1]);
            int year = Integer.parseInt(aDate[2]);

            String[] aTime = time.replace(" ", ":").split(":");
            int hour = Integer.parseInt(aTime[0]);
            int hourCopy = hour;
            int minute = Integer.parseInt(aTime[1]);
            String APM = (aTime[2]);

            if (APM.equals("PM")) {
                    hourCopy += 12;
            }
  }
```

The first couple of lines of this function assign the values that we put in the appropriate fields of the Event Activity (for example, our "course" that we type in is stored in the variable course). The next block of code starting from `(if mRowId == null)` does the following: if the rowId we set comes out to null, set it to something and then call the `createReminder()` function in `mDbHelper`. Otherwise, call the `updateReminder()` function in `mDbHelper`. Let's take a look at what these functions do by opening up the *HomeworkDbAdapter.java* file.

The `createReminder()` and `updateReminder()` functions accept the same kind of information (all the data that we enter into each of the inputs in the Event Activity). The difference is that we also pass in a rowID when we are updating reminders.

Let's try running our application and this time, trying entering some information into each of the fields and then pressing the "Save Changes" button. If you get back to the Main Activity -- great! Your Main Activity should look like Figure 5.12 below:

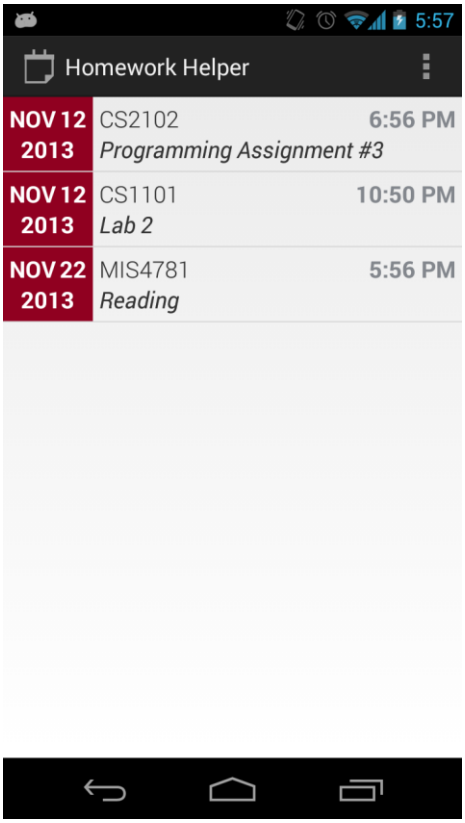

Figure 5.12: ListView of reminders

Take a look at the *activity_main.xml* file in the *res* → *layout* folder:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:tools="http://schemas.android.com/tools"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:orientation="vertical" >

      <ListView android:id="@android:id/list"
      android:layout_width="match_parent"
      android:layout_height="match_parent"/>

      <TextView android:id="@android:id/empty"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
```

```
        android:text="@string/main_no_reminders"
        android:gravity="center"
        android:textSize="18sp"/>
</RelativeLayout>
```

We already made the user interface for you and you have already seen it in action. When there aren't any reminders, you get a centered **TextView** that says "You don't have any reminders", otherwise you get a **ListView.** You might be wondering how we set the different colors and fonts for each reminder that you set. Let's take a look at *reminders_row.xml*. Here we set a bunch of **TextViews** and gave them specific IDs so that we can refer to them later in *MainActivity.java*. Each time you create a reminder, a new *reminders_row* gets added to the **ListView** in the previous XML file.

Going back to the application, the idea is to tap on a reminder that you set so that you can do two things:

- edit the reminder
- delete the reminder

So if you currently tap on a reminder, what happens? You get a screen similar to Figure 5.13 below:

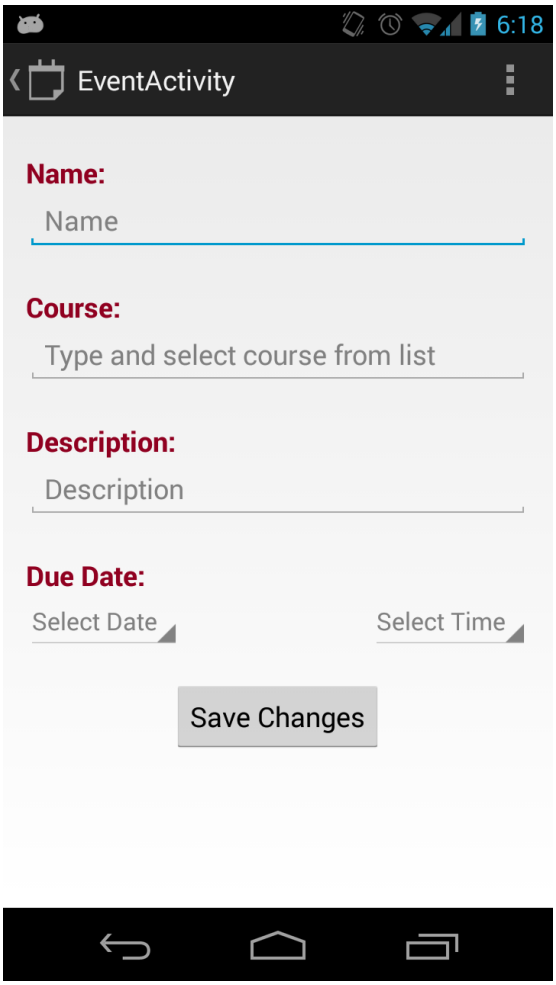

Figure 5.13: Editing the Event Activity

In theory, tapping on the reminder should automatically populate the appropriate fields in the Event Activity -- right? Let's fix that right now by creating a function called `populateFields()` after the `onCreateOptionsMenu()` method:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  // Inflate the menu; this adds items to the action bar if it is present.
  getMenuInflater().inflate(R.menu.event, menu);
  return true;
}

private void populateFields() {
      if (mRowId != null) {
        Cursor reminder = mDbHelper.fetchReminder(mRowId);
        startManagingCursor(reminder);
        mNameText.setText(reminder.getString(

      reminder.getColumnIndexOrThrow(HomeworkDbAdapter.KEY_NAME)));
```

```
        mCoursesText.setText(reminder.getString(reminder.getColumnIndexOrThrow(
HomeworkDbAdapter.KEY_COURSE)));

mDescriptionText.setText(reminder.getString(
reminder.getColumnIndexOrThrow(HomeworkDbAdapter.KEY_DESCRIPTION)));

mDateText.setText(reminder.getString(reminder.getColumnIndexOrThrow(HomeworkD
bAdapter.KEY_DUE_DATE)));


mTimeText.setText(reminder.getString(reminder.getColumnIndexOrThrow(HomeworkD
bAdapter.KEY_DUE_TIME)));

setTitle(mCoursesText.getText().toString() + " - " +
mNameText.getText().toString());
        }
}
```

Then be sure to add the following function call to the `onCreate()` function right before the `mDateText` listener:

```
        // Apply the adapter to the AutoCompleteTextView
        mCoursesText.setAdapter(adapter);

        populateFields();

        // Listener for 'Set Date' TextView
        mDateText.setOnClickListener(new View.OnClickListener() {
        …
```

This function we just created retrieves the values from a specific row of our SQL database and uses the `setText().toString()` function to do so. To better visualize what our SQL database looks like and how things are stored, take a look at Figure 5.14 and Figure 5.15 below:

_id	Name	Course	Description	Date	Time
1	Lab 2	CS1101	Do something in Dr. Racket	11/25/ 2013	7:00 PM

Figure 5.14: Visual representation of SQL table for HomeworkHelper

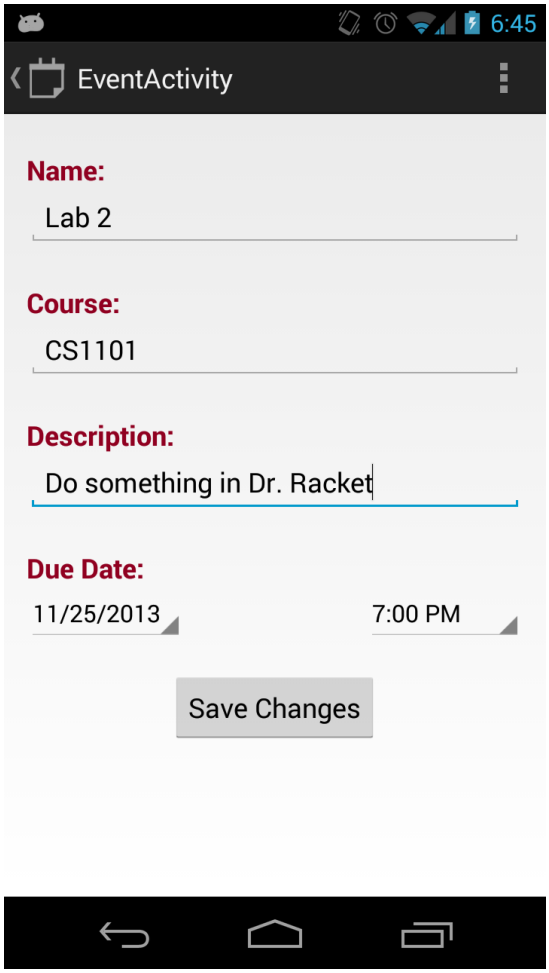

Figure 5.15: How the Event Activity retrieves data from SQL table

There are still a few things we have to configure in *EventActivity.java*. Let's start by adding these functions under the one we just configured:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putSerializable(HomeworkDbAdapter.KEY_ROWID, mRowId);
}

@Override
protected void onPause() {
    super.onPause();
}

@Override
protected void onResume() {
    super.onResume();
    populateFields();
```

```
    }
```

The functions above are needed for the Activity lifecycle of the Event Activity. This was already explained in the Stub App. One last thing we need to configure is the `requestCode` we talked about earlier in the chapter. Right now, the `populateFields()` function is not executing because `requestCode` is null – which is another term for "nothing". Let's add the following block of code right after the `populateFields()` function call in the `onCreate()` method:

```
...
populateFields();

requestCode = getIntent().getIntExtra(MainActivity.REQUEST_CODE, -1);
if (requestCode == 0) {
   setTitle(R.string.action_create);
}

if (requestCode == 1) {
   mRowId = (savedInstanceState == null) ? null :
(Long) savedInstanceState.getSerializable(HomeworkDbAdapter.KEY_ROWID);
   if (mRowId == null) {
      Bundle extras = getIntent().getExtras();
      mRowId = extras != null ? extras.getLong(HomeworkDbAdapter.KEY_ROWID):
null; }
   else {
      mRowId = null;
   }
}
```

Now what is this `requestCode` exactly? This is an integer with a value of either 0 or 1 that we use to tell our application when we want to either create a new reminder, or update an existing reminder. This prevents us from having to make a completely new Activity that looks exactly like the Event Activity. Take a look at the *MainActivity.java* file and look for the `createReminder()` function and the `onListClickItem()` function:

```
...
public void createReminder() {
       Intent i = new Intent(this, EventActivity.class);
       i.putExtra(REQUEST_CODE, ACTIVITY_CREATE);
       startActivityForResult(i, ACTIVITY_CREATE);
}

@Override
protected void onListItemClick(ListView l, View v, int position, long id) {
      super.onListItemClick(l, v, position, id);
      Intent i = new Intent(this, EventActivity.class);

      i.putExtra(HomeworkDbAdapter.KEY_ROWID, id);
```

```
        i.putExtra(REQUEST_CODE, ACTIVITY_EDIT);
        startActivityForResult(i, ACTIVITY_EDIT);
}
```

Remember that we call the function `createReminder()` when we tap on the "Create Reminder" menu item we set in *MainActivity.java.* The `onListItemClick()` function is called when a user taps on a reminder that they set in *MainActivity.java*. In both functions, we use an Intent *i* to go to the Event Activity. The `putExtra()` function is used here to allow us to send any additional information we want. In this case, ACTIVITY_CREATE is equal to 0 and ACTIVITY_EDIT is equal to 1. We then use the `startActivityForResult()` function using the Intent *i,* and the integer.

Take a look at the code we recently put into *EventActivity.java*. You'll notice that if the `requestCode` it receives is 0, we set the Activity title using the `setTitle()` function. Otherwise, we obtain the `mRowID` of the reminder we set from our SQL table and set it so that it retrieves the values we need in the `populateFields()` function.

Now try running the application. You should now be able to create new reminders and edit them by tapping on the one you want in the Main Activity! Test your application by making sure you can do this as many times as you want. After you do this several times, you may notice that you can get away with doing something.

Did you notice the bug? Load up the application and select the "Create Reminder" option. Now don't put anything into any of the fields for Name, Course, Description, etc. Just go right ahead and tap on the "Save Changes" button. What happens next? This time, create a reminder but only set the Time and Date -- nothing else. See what happens? You should see something similar in either one of the screenshots of Figure 5.16 below:

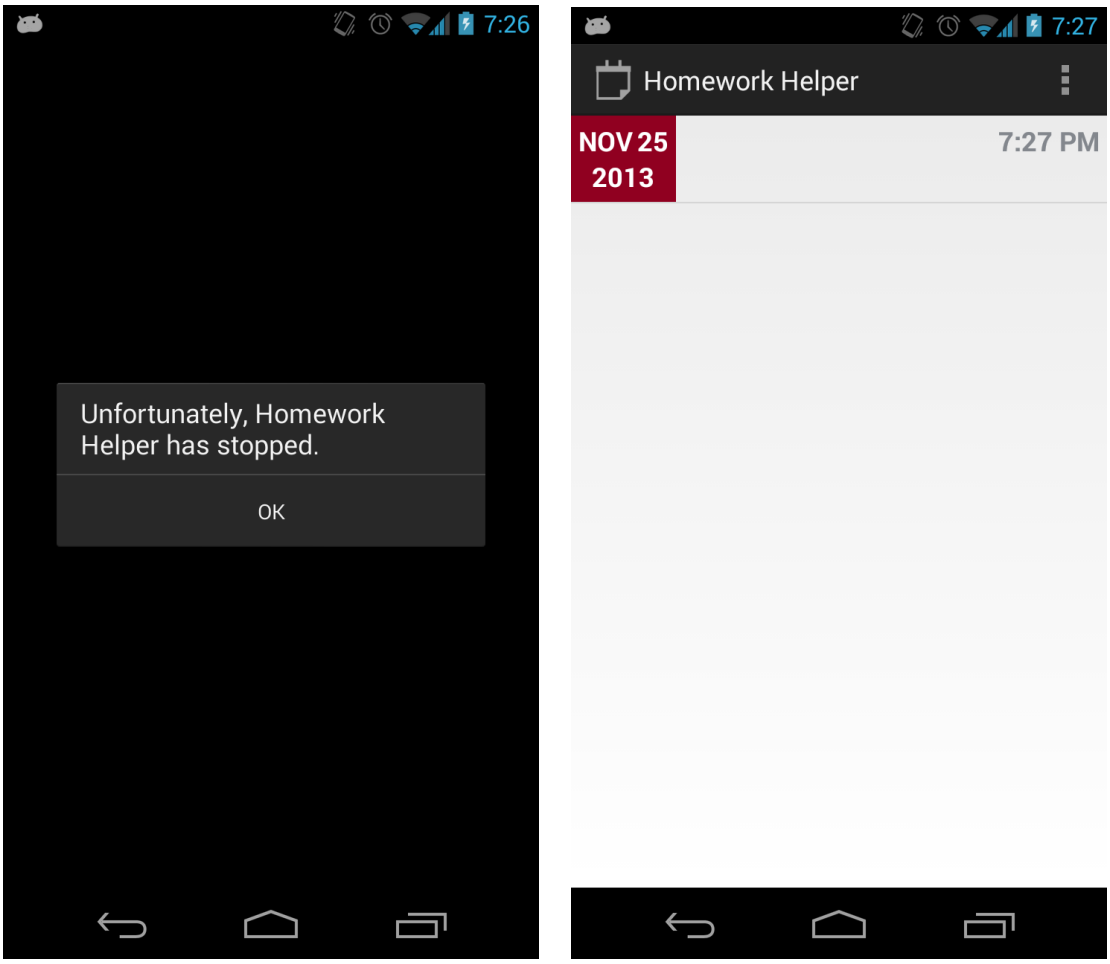

Figure 5.16: HomeworkHelper Bugs

In both instances, this is occurring because we are leaving things blank. We should be checking to make sure that what a user types in is valid input. This process known as *validation* is highly essential when making applications such as these. Fortunately, Android already has a built-in feature that lets us simplify this process. Let's fix this bug by opening up *EventActivity.java* and modifying our event listener for the "Save Changes" button:

```
mSaveChanges.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
            if (mNameText.getText().toString().length() == 0) {
                  mNameText.setError("Reminder name is required!");
                  return;
            }
            else if (mCoursesText.getText().toString().length() == 0) {
                  mCoursesText.setError("Select one of the available
courses!");
                  return;
            }
            else if (mDescriptionText.getText().toString().length() == 0) {
```

```
                mDescriptionText.setError("Reminder description is
required!");
                return;
            }
            else if (mDateText.getText().toString().length() == 0) {
                mDateText.setError("Reminder date is required!");
                return;
            }
            else if (mTimeText.getText().toString().length() == 0) {
                mTimeText.setError("Reminder time is required!");
                return;
            }
            else {
                saveState();
                setResult(RESULT_OK);
                finish();
            }
```

We can use the built in `setError()` function in Android to allow us to validate user input as shown in the code above and to give a customized error message for each field. We write several if-statements to check whether or not each field is empty using the `length()` function. The “Save Changes” button won’t work unless *all* fields have some value in them. Figure 5.17 below shows us what happens if a field is left blank:

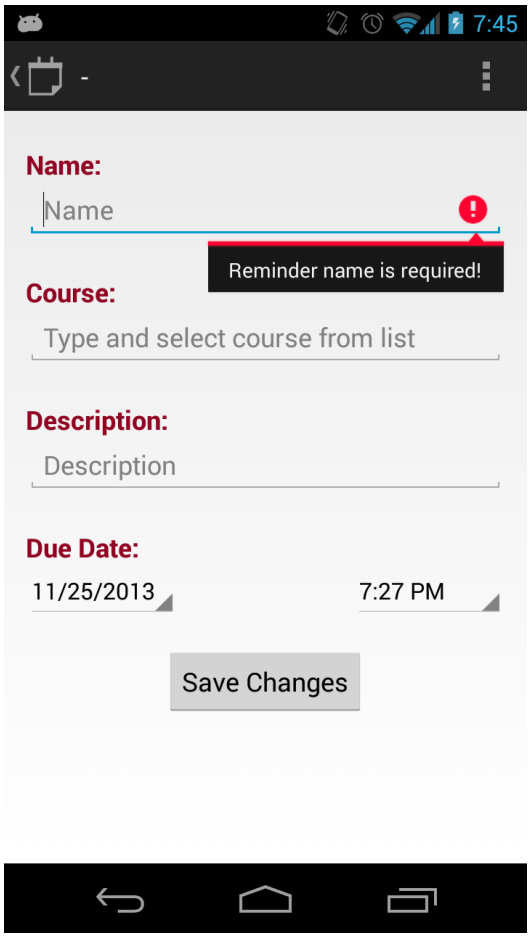

Figure 5.17: Example of input validation using setError()

Save these changes and run your application again. If you aren't able to create a blank reminder you have successfully fixed the bug! One thing you might have noticed by now, we can create and edit reminders… but how do we *delete* them in case we don't want them anymore? Let's add this code in *EventActivity.java* right after the `setContentView()` function call in the `onCreate()` method:

```
...
setContentView(R.layout.activity_event);

ActionBar actionBar = getActionBar();
actionBar.setDisplayHomeAsUpEnabled(true);


mDbHelper = new HomeworkDbAdapter(this);
...
```

Import any necessary packages and save changes. If prompted for a specific package, be sure to select **android.app.ActionBar**.

The code we just entered declares something called an **ActionBar**, which is the header of our application. We will be adding a delete button to this **ActionBar** so that we can delete reminders.

Now open up the *event.xml* file in the *res* → *menu* folder. Delete the following code and replace it with this:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
~~<item~~
~~android:id="@+id/action_settings"~~
~~android:orderInCategory="100"~~
~~android:showAsAction="never"~~
~~android:title="@string/action_settings"/>~~

    <item android:id="@+id/action_discard"
        android:icon="@drawable/ic_action_discard"
        android:title="@string/action_discard"
        android:showAsAction="ifRoom" />
</menu>
```

The code above replaced a menu item with an **ActionBar** item with an icon that we put for you called *action_discard*. If you try running your application at this point, you'll notice that you now have a button in the **ActionBar** but you cannot do anything with it just yet. When we created the menu items for the Main Activity, we used a function called `onOptionsItemSelected()` to do something. We're going to do something very similar in *EventActivity.java* so add this code right after the `onResume()` function:

```
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle presses on the action bar items
        switch (item.getItemId()) {
            case R.id.action_discard:
                // Delete event
                AlertDialog.Builder alertDialogBuilder = new
AlertDialog.Builder(this, 2);

                alertDialogBuilder.setTitle(R.string.action_discard);

                alertDialogBuilder.setMessage
(R.string.action_discard_question);
                alertDialogBuilder.setCancelable(false); // Sets whether this
dialog is cancelable with the BACK key.

                // "Yes, delete this reminder"
                alertDialogBuilder.setPositiveButton("Yes",new
DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog,int id) {
```

```
                        mDbHelper.deleteReminder(mRowId);
                        finish();
                    }
                });
                // "No, I changed my mind"
                alertDialogBuilder.setNegativeButton("No",new
DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog,int id) {
                        dialog.cancel();
                    }
                });

                AlertDialog alertDialog = alertDialogBuilder.create();
                alertDialog.show();

                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }
```

Import any necessary packages and save your changes. Similar to the QuizMe application, the code above builds an **AlertDialog** when the discard button is tapped on, giving the user a confirmation dialog box that asks if they are sure they want to delete the reminder. Try running the application. You should now be able to delete reminders as well! There is one thing that we have to fix here, and that is that you should only be able to delete *existing* reminders and not new ones (because they are not in the database yet) -- which could cause errors. Let's fix this by modifying the `onCreateOptionsMenu()` function in *EventActivity.java*:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  // Inflate the menu; this adds items to the action bar if it is present.
  if (requestCode == 0) getMenuInflater().inflate(R.menu.event, menu);
  return true;
}
```

The code above fixes the bug we mentioned by only displaying the button if we are creating a new reminder. Therefore, requestCode is equal to 0. Otherwise, the button does not show up and the user is not able to delete a new reminder that hasn't been made yet.

Feel free to test your application and let's move on to creating a Settings Activity.

5.5: Settings Activity

Overview

At this point, most of our application is done, yet there is no way that a student can configure what they want from the application. For example, what if a user does not want our

application to always load a Splash Screen at startup? How do we go about asking for a preference? For this, we will be adding a Settings Activity a very common feature in many Android applications. The Settings Activity is exactly what it sounds like: a menu allowing users to configure various options of the application to suit their needs.

Development

Let's start by making a new Activity. Instead of doing this the way we did in the previous sections, start by right-clicking the *src* folder in the *HomeworkHelper* project and selecting **New → Class**. Fill out the information as shown in Figure 5.18 below and then click the **Finish** button:

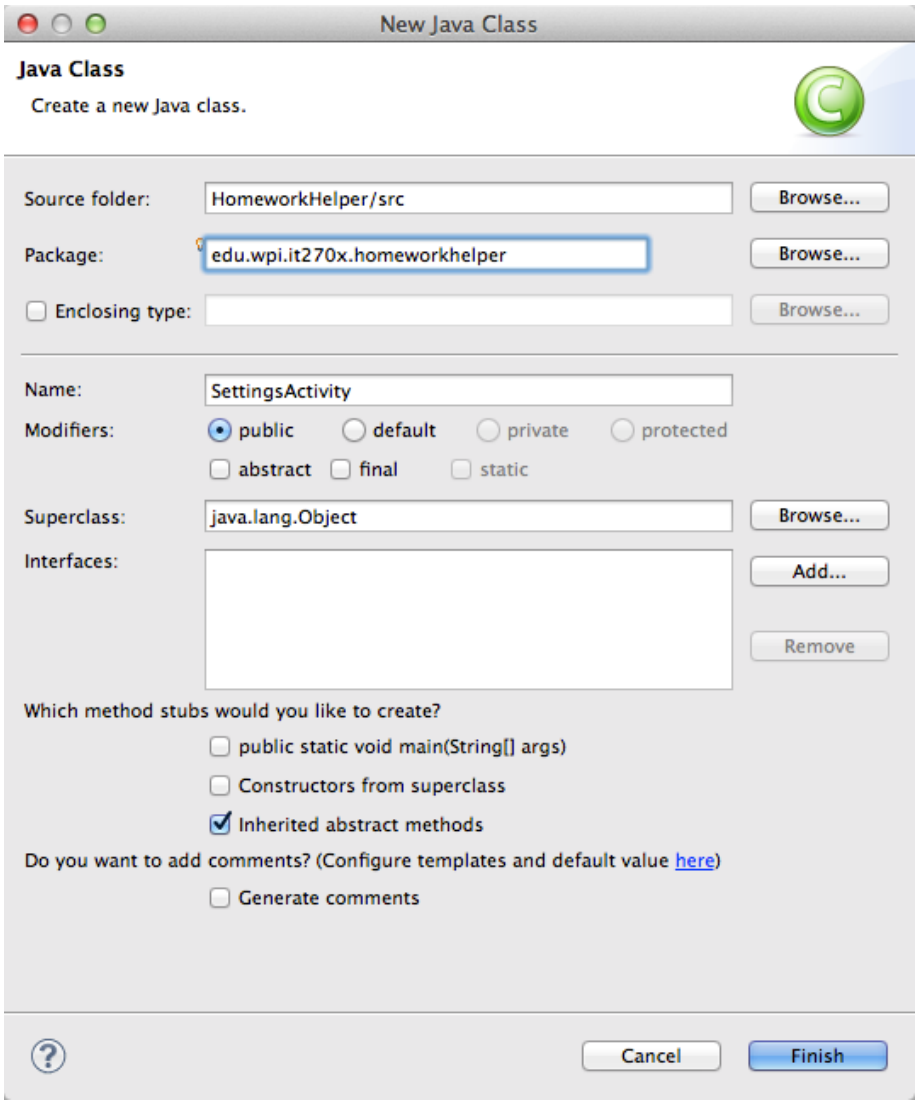

Figure 5.18: Creating the Settings Activity

When the file loads up, replace everything in it with this block of code:

```
package edu.wpi.it270x.homeworkhelper;

public class SettingsActivity extends PreferenceActivity {
	@Override
	public void onCreate(Bundle savedInstanceState) {
	    super.onCreate(savedInstanceState);
	    addPreferencesFromResource(R.xml.preferences);
	}

}
```

Import any necessary packages and then save your changes. You will notice that `R.xml.preferences` remains underlined. That's because we haven't made this file yet. Let's do that by right-clicking the *res* → *xml* folder and selecting **New** → **Android XML File** as shown in Figure 5.19 below:

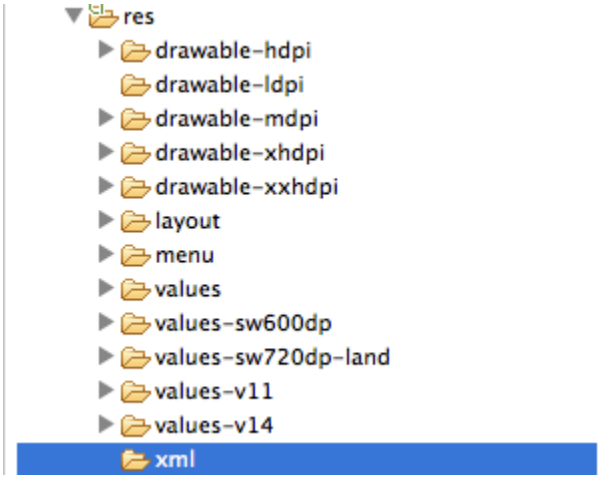

Figure 5.19: XML folder

You should then see a window similar to Figure 5.20 below. Be sure that **Preference** is selected next to *Resource Type* and that the file name is *preferences* as shown then click the **Finish** button:

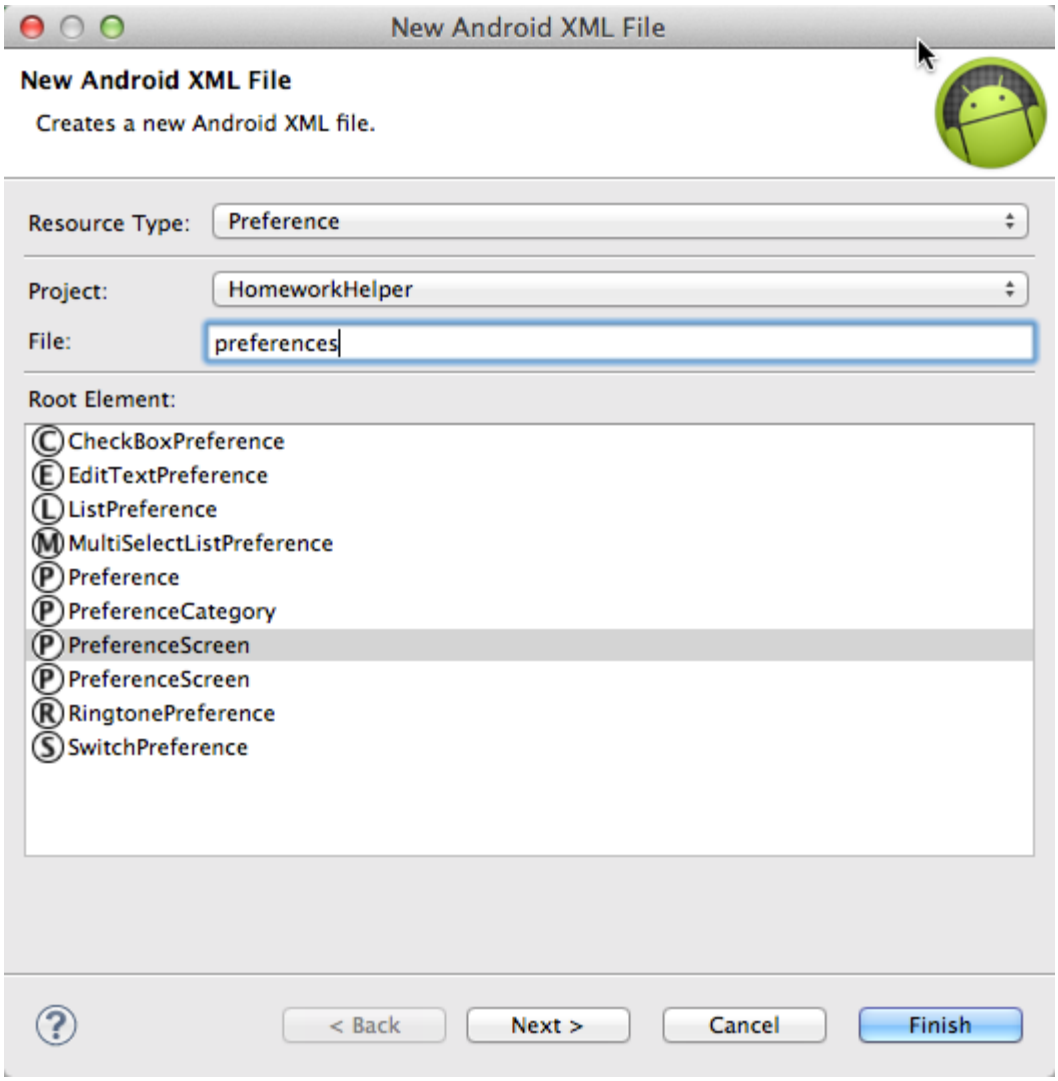

Figure 5.20: Creating the preferences.xml file

When the file loads up, replace everything in it with the following lines of code:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android"
>

    <PreferenceCategory
        android:key="pref_key_storage_settings"
        android:title="@string/pref_general_settings_header" >
        <CheckBoxPreference
            android:id="@+id/isCalendarEnabled"
            android:defaultValue="false"
            android:key="isCalendarEnabled"
            android:summary="@string/pref_calendar_summary"
            android:title="@string/pref_calendar_title" />
```

```
        <CheckBoxPreference
            android:id="@+id/isNotificationEnabled"
            android:defaultValue="false"
            android:key="isNotificationEnabled"
            android:summary="@string/pref_notifications_summary"
            android:title="@string/pref_notifications_title" />
        <CheckBoxPreference
            android:id="@+id/isSplashEnabled"
            android:defaultValue="true"
            android:key="isSplashEnabled"
            android:summary="@string/pref_splash_summary"
            android:title="@string/pref_splash_title" />
        <CheckBoxPreference
            android:id="@+id/isAutoDeleteEnabled"
            android:defaultValue="false"
            android:key="isAutoDeleteEnabled"
            android:summary="@string/pref_past_reminders_summary"
            android:title="@string/pref_past_reminders_title" />
    </PreferenceCategory>

</PreferenceScreen>
```

Save all changes and you should not have any more errors at this point. Now let's add the following block of code to *MainActivity.java* right after the first `case` statement in the `onOptionsItemSelected()` function:

```
      ...
      case R.id.action_create:
            createReminder();
            return true;
      case R.id.action_settings:
            goToSettings();
            return true;
      ...
```

Save all changes. You will notice that the function call `goToSettings()` remains underlined because we haven't defined it yet in *MainActivity.java.* Therefore, let's add the following block of code after the `createReminder()` function:

```
...
public void createReminder() {
       Intent i = new Intent(this, EventActivity.class);
       i.putExtra(REQUEST_CODE, ACTIVITY_CREATE);
       startActivityForResult(i, ACTIVITY_CREATE);
}

public void goToSettings() {
        Intent intent = new Intent(this, SettingsActivity.class);
        startActivity(intent);
}
```

Now open up your *AndroidManifest.XML* file and add the following code right below the Event Activity:

```
<!-- Event Activity -->
        <activity
            android:name=".EventActivity"
            android:label="@string/title_activity_event"
            android:parentActivityName=".MainActivity" >
            <meta-data
                android:name="android.support.PARENT_ACTIVITY"
                android:value=".MainActivity" />
        </activity>

 <!-- Settings Activity -->
        <activity
            android:name=".SettingsActivity"
            android:theme="@android:style/Theme.Holo.NoActionBar" >
        </activity>
...
```

Save all your changes and now you should not have any more errors at this point. Run the application and instead of creating a reminder, tap on the Settings menu item. You should see something similar to Figure 5.21 below:

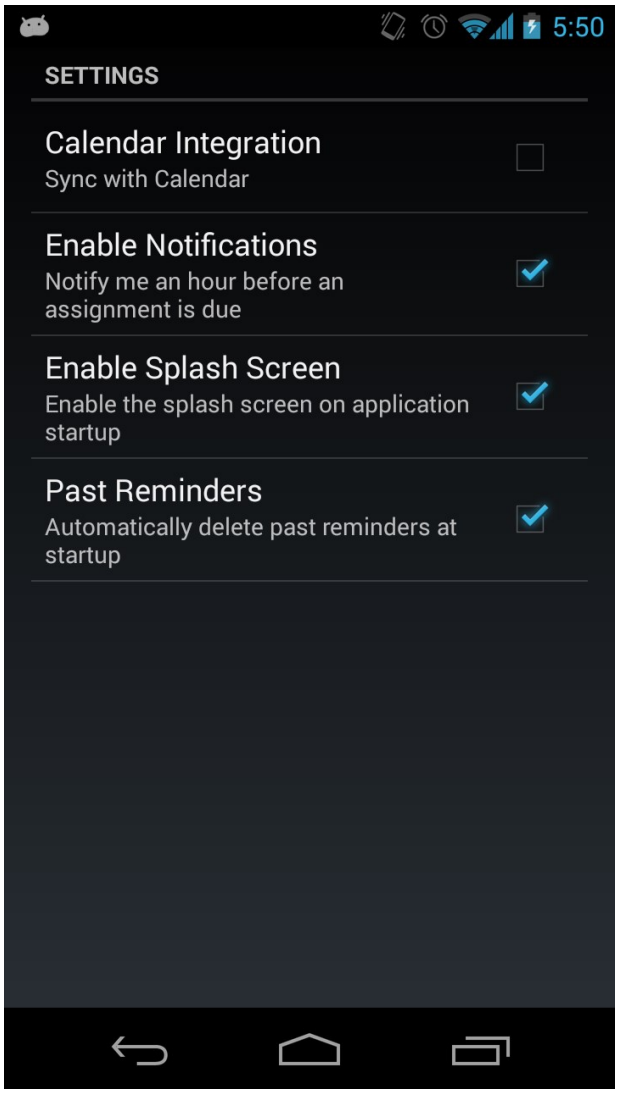

Figure 5.21: Settings Activity Layout

Now take a look back at the *preferences.xml* file we made earlier. This will be much easier to visualize if you have both the application running and the file open. Each of the four items that you see can be defined as **CheckBoxPreferences**. Each of these has an id, default value, key, summary, and a title. With these checkboxes, preferences are stored in the key attribute as *booleans* -- True or False. Right now, Figure 5.22 shows each preference with a value of True (meaning that they are checked). All these preferences are grouped using **PreferenceCategory** with the header 'Settings' as shown in the figure above.

Now how do we retrieve what these values are across all the different activities that we have? Let's start by allowing a user to enable or disable the Splash Screen. Open up the *Splashscreen.java* file and modify the following code after the `super.onResume()` function call in the `onResume()` method:

```
@Override
```

```
    protected void onResume()
    {
        super.onResume();
        SharedPreferences sp =
PreferenceManager.getDefaultSharedPreferences(this);
        // Obtain the sharedPreference, default to true if not available
        boolean isSplashEnabled = sp.getBoolean("isSplashEnabled", true);

        if (isSplashEnabled)
        {
            new Handler().postDelayed(new Runnable()
            {
                @Override
                public void run()
                {
                    //Finish the splash activity so it cannot be returned to.
                    finish();

                    // Create an Intent that will start the Main Activity.
                    Intent mainIntent = new Intent(Splashscreen.this,
MainActivity.class);
                    startActivity(mainIntent);
                }
            }, SPLASH_TIME_OUT);
        }
        else
        {
            // if the splash is not enabled, then finish the activity
immediately and go to main.
            finish();
            Intent mainIntent = new Intent(this, MainActivity.class);
            startActivity(mainIntent);
        }
}
```

Import any necessary packages and then save your changes. Notice, at the start of the function we initialized a *SharedPreferences* object and get the value (stored as the key attribute `isSplashEnabled` from the *preferences.xml* file). If it is True, show the Splash Screen. Otherwise, go right into the Main Activity. Test this out by running the application and enabling/disabling that checkbox in the *Settings*.

Once you have done that, let's use this in *MainActivity.java* to allow us to automatically delete past reminders on the startup of the application. Open up this file and add the following code:

```
...
public static final String REQUEST_CODE = "_requestCode";
private SharedPreferences sp;
```

```
...
@Override
protected void onCreate(Bundle savedInstanceState) {
        sp = PreferenceManager.getDefaultSharedPreferences(this);
        boolean isAutoDeleteEnabled = sp.getBoolean("isAutoDeleteEnabled",
false);

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mDbHelper = new HomeworkDbAdapter(this);
        if (isAutoDeleteEnabled) {
           ArrayList<Integer> oldRows = mDbHelper.fetchOldReminders();

           if (oldRows.size() != 0) {
               for (int x = 0; x < oldRows.size(); x++) {
                   mDbHelper.deleteReminder(oldRows.get(x));
               }

               AlertDialog.Builder alertDialogBuilder = new
AlertDialog.Builder(this, 2);


               alertDialogBuilder.setTitle(R.string.main_deleted_reminders);
               alertDialogBuilder.setMessage("Removed " + oldRows.size() + "
reminders");
               alertDialogBuilder.setCancelable(true); // Sets whether this
dialog is cancelable with the BACK key.

               AlertDialog alertDialog = alertDialogBuilder.create();
               alertDialog.show();
           }
        }
        fillData();
...
```

Import the necessary packages and save all your changes. Notice that the function above finds out whether or not the `isAutoDeleteEnabled` key is True or False. If it is True (meaning that it is checked in the Settings Activity*)* then use the already existing `deleteReminder()` function to delete old reminders. Note, we created a database function to tell us what old reminders are. If you'd like, look for the `fetchOldReminders()` function in the *HomeworkDbAdapter.java* file for more information. Feel free to run the application and test those two checkboxes in the Settings Activity.

5.6: Notifications and Calendar Support

Overview

Our application is almost completely finished. We just have a few more final touches that we have to add before we consider this project complete. First of all, it would be extremely helpful if this application notified you when a due date was approaching, rather than having to constantly check the application. In addition, the Android operating system already comes with a robust Calendar application installed. Integrating Homework Helper with this application (so that our reminders are placed into our Calendar automatically) would also be very helpful.

Development

Our first step is to allow students to automatically send reminders to their built-in Calendar application on their Android Devices. To do this, open up *EventActivity.java* and modify the following in the `saveState()` function:

```
  ...
  if (APM.equals("PM")) {
      hourCopy += 12;
  }
  sp = PreferenceManager.getDefaultSharedPreferences(this);
  boolean isNotificationEnabled = sp.getBoolean("isNotificationEnabled",
false);
  boolean isCalendarEnabled = sp.getBoolean("isCalendarEnabled", false);


  if (isCalendarEnabled) {

      Intent calIntent = new Intent(Intent.ACTION_EDIT);
      Calendar startTime = Calendar.getInstance();
      startTime.set(year, month, day, hourCopy, minute);

      if (requestCode == 0) {
          calIntent.setData(Events.CONTENT_URI);
          calIntent.putExtra(Events._ID, mRowId);
          calIntent.putExtra(Events.TITLE, title);
          calIntent.putExtra(Events.EVENT_LOCATION, course);
          calIntent.putExtra(Events.DESCRIPTION, description);

          calIntent.putExtra(CalendarContract.EXTRA_EVENT_BEGIN_TIME,
startTime.getTimeInMillis());
          startActivity(calIntent);
      }
```

Be sure to include the following with all the other variable declarations at the top of the file:

```
private int requestCode;
SharedPreferences sp;
```

Import any required packages and save all your changes. In short, the code we just added retrieves all the values a user inputs in the Event Activity and then opens up the Calendar application with all the appropriate fields already filled in depending on whether or not this setting is enabled/disabled. Try running the application and test this for yourself. You'll notice that we only configured this for *new* reminders and not existing ones. Therefore, once you add a reminder to your calendar, you can't edit it unless you do so using the Calendar application. This is something you may want to figure out how to change.

The last step of the project is to allow support for notifications. Let's begin by right clicking the *src* folder in the *HomeworkHelper* project and selecting **New** → **Class**. Name this file "NotificationService" and be sure that the package selected is **edu.wpi.it270x.homeworkhelper** then press the **Finish** button. Once the file opens, replace everything in it with this code:

```
package edu.wpi.it270x.homeworkhelper;

public class NotificationService extends IntentService {

	private static final String TAG = "NotificationService";

	public NotificationService() {
	    super(TAG);
	}

	@Override
	protected void onHandleIntent(Intent intent) {
	    Log.i(TAG, "Received as intent: " + intent);

	    Intent notificationIntent = new Intent (this, MainActivity.class);
	    PendingIntent pendingIntent = PendingIntent.getActivity(this, 0,
notificationIntent, 0);

	    Resources r = getResources();

	    Notification notification = new NotificationCompat.Builder(this)
	            .setTicker(r.getString(R.string.notification_new))
	            .setSmallIcon(R.drawable.ic_launcher)
	            .setContentTitle(intent.getStringExtra("_course") + " - " +
intent.getStringExtra("_name"))
	            .setContentText(intent.getStringExtra("_description"))
	            .setContentInfo("Due: " + intent.getStringExtra("_time"))
	            .setContentIntent(pendingIntent)
	            .setAutoCancel(true)
	            .setLights(Color.GRAY, 3000, 3000)

.setSound(RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION))
	            .build();
```

```
            NotificationManager notificationManager = (NotificationManager)
                    getSystemService(NOTIFICATION_SERVICE);

            int rowId = (int) intent.getLongExtra("_rowId", 0);
            notificationManager.notify(rowId, notification);

      }

      public static void setServiceAlarm(Context context, boolean isOn, long
rowId, String name, String course, String description, int year, int month,
int day, int hour, int minute, String time) {
            Intent i = new Intent(context, NotificationService.class);
            i.putExtra("_rowId", rowId);
            i.putExtra("_name", name);
            i.putExtra("_course", course);
            i.putExtra("_description", description);
            i.putExtra("_year", year);
            i.putExtra("_month", month);
            i.putExtra("_day", day);
            i.putExtra("_hour", hour);
            i.putExtra("_minute", minute);
            i.putExtra("_time", time);

            PendingIntent pi = PendingIntent.getService(
                      context, (int)rowID, i,
PendingIntent.FLAG_UPDATE_CURRENT);

            AlarmManager alarmManager = (AlarmManager)
                      context.getSystemService(Context.ALARM_SERVICE);

            if (isOn) {
                Calendar startTime = Calendar.getInstance();
                startTime.set(year, month, day, hour, minute);

                alarmManager.set(AlarmManager.RTC_WAKEUP,
                          startTime.getTimeInMillis() - (1000 * 60 * 60),
pi);
            } else {
                alarmManager.cancel(pi);
                pi.cancel();
            }
      }

      public static boolean isServiceAlarmOn(Context context) {
            Intent i = new Intent(context, EventActivity.class);
            PendingIntent pi = PendingIntent.getService(
                       context, 0, i, PendingIntent.FLAG_NO_CREATE);
            return pi != null;
```

```
        }

}
```

Import any missing packages. If prompted, select the **android.util.Log** import when importing the *Log* function. Save all changes and add the following code under the code we last added in *EventActivity.java*:

```
…
if (isNotificationEnabled) {
  boolean shouldStartAlarm = !NotificationService.isServiceAlarmOn(this);
  NotificationService.setServiceAlarm(this, shouldStartAlarm, mRowId, title,
course, description, year, month, day, hourCopy, minute, time);
}
```

The code above calls a function made in *NotificationService.java* that sends all of our inputted data to `setServiceAlarm()`. In this function, we use something called an **AlarmManager** to wake up the CPU of the Android Device at a specific time. Here, we set it so that it does this approximately an hour before the time you set when using the **TimePickerFragment** in the Event Activity.

NotificationService.java is something a little different than an Activity since it runs in the background and doesn't require the application to be running. If you force close the application you also close this *Service.*

Don't forget to add the following code to the *AndroidManifest.xml* file underneath the Settings Activity:

```
<!-- Settings Activity -->
  <activity
    android:name=".SettingsActivity"
    android:theme="@android:style/Theme.Holo.NoActionBar" >
  </activity>

<!-- Notification Service -->
   <service android:name=".NotificationService"/>
```

Save your changes and try running the application. First make sure that the Notification setting is *enabled*. Then, make a new reminder and set it for approximately one hour from now. If all goes well, you should see something similar to Figure 5.22 below:

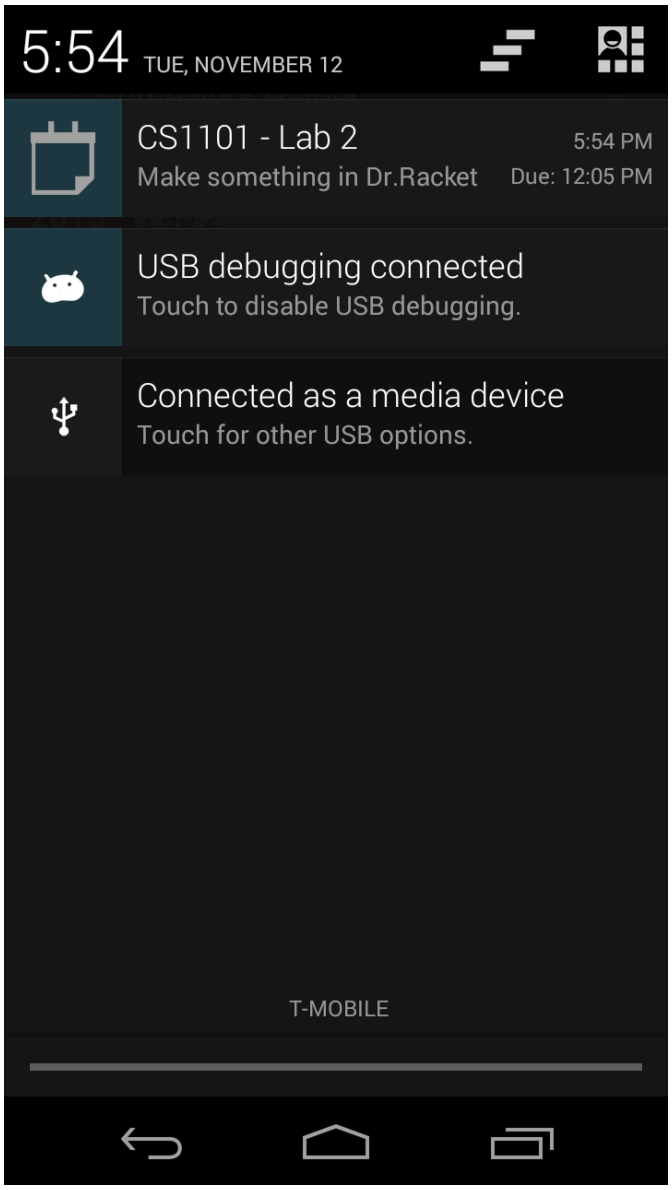

Figure 5.22: Notification from HomeworkHelper

If you look back at the code we set, we use a *builder* similar to the **AlertDialog** we used previously to create the details of the reminder for each notification. In addition, tapping on the notification takes you right to the Main Activity of the application. This is set using the following code in *NotificationService.java* in the `onHandleIntent()` function:

```
Intent notificationIntent = new Intent (this, MainActivity.class);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0,
notificationIntent, 0);
```

Congratulations! You have successfully completed the HomeworkHelper application.

5.7: Additional Functionality

Now that the application is complete, what have you learned across the way? How would you expand this project with additional functionality? We thought of a few ideas:

- The SQLite database is stored on the Android Device by default. Is it possible to have it stored elsewhere - say online? What would be the advantages/disadvantages of doing this?
- If a student really uses this application to its fullest, what kind of additional features would they like to see? Here are some of our thoughts:
    - Be able to search through reminders in the Main Activity based off of any matching results
    - Be able to categorize all your reminders by month. For example, have a drop down list somewhere in the Main Activity that only displays reminders by the month/year selected
    - Delete multiple reminders directly from the Main Activity without having to go through each reminder individually. For example, tap and hold a reminder and repeat the process for any other reminders which opens up a dialog asking to delete all the selected reminders

Run the entire application several times across several Android devices. Do any bugs stick out the most? How would you fix these?

Conclusion

If you've made it this far, feel free to congratulate yourself. Throughout this entire project, our team spent many hours both in development and in researching through a multitude of online resources to be able to teach prospective students the necessary skills of Android application development. Mobile applications continue to be a growing market and having the skill to develop these applications will prove to be highly beneficial.

If you've become hooked on making Android applications just as much as we have, we highly encourage you to continue developing and learning more on the subject. We definitely could not have done this project without hours of research, collaboration, and dedication. You would be surprised at how much useful documentation you can find for nearly any subject of application development, and even for programming in general. As well, there are many excellent communities out there willing to help other developers create the best products they can, and we highly suggest getting involved with some of these communities.

Try getting involved with developing existing applications or making some from the ground up like we did. You can find many open source applications on GitHub (https://github.com) that you can not only use but also contribute to.

As an individual, you can only accomplish so much. However, working as a team with other developers allows you to write clean, bug-free, and efficient code to make useful and robust applications that many other people can use. If you're up for the challenge, try making an application for the Google Play store and watch what happens next.

Works Cited

Google Inc. "Develop | Android Developers." *Android Developers*. Google Inc., n.d. Web. 21 Nov. 2013. <http://developer.android.com/develop/index.html>.

Phillips, Bill, and Brian Hardy. *Android Programming: The Big Nerd Ranch Guide*. N.p.: Big Nerd Ranch Guides, 2013. *Safari Books Online*. Web. <http://proquest.safaribooksonline.com/9780132869126>.

Tamada, Ravi. "Android SQLite Database Tutorial." *Android SQLite Database Tutorial*. N.p., 27 Nov. 2011. Web. 21 Nov. 2013. <http://www.Androidhive.info/2011/11/android-sqlite-database-tutorial/>.